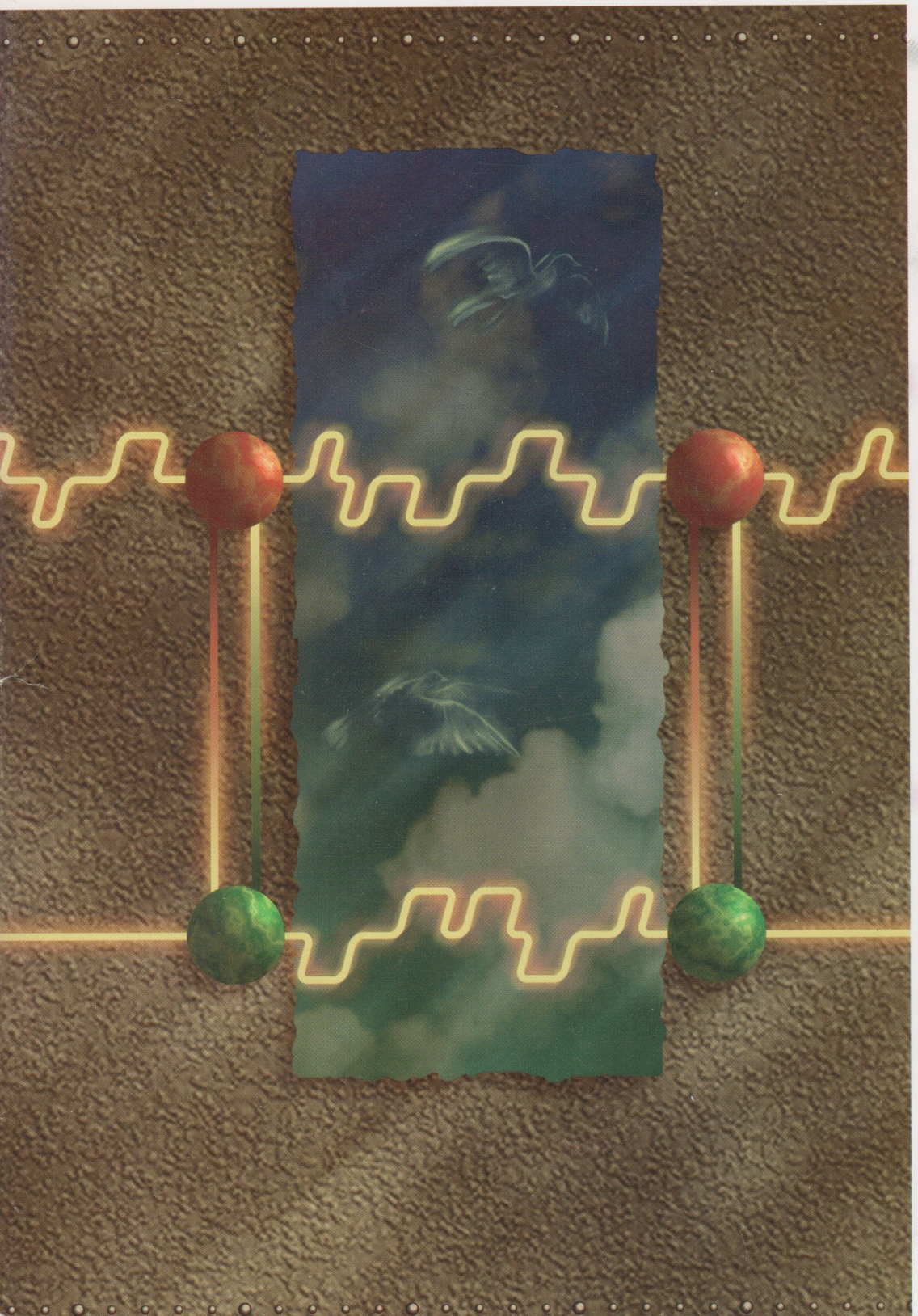
develop

The Apple Technical Journal



ASYNCHRONOUS ROUTINES ON THE MACINTOSH

INSIDE QUICKTIME AND COMPONENT-BASED MANAGERS

MACINTOSH DEBUGGING: THE BELLY OF THE BEAST REVISITED

ADVENTURES IN COLOR PRINTING

DEVICELOOP MEETS THE INTERFACE DESIGNER

LOOKING AHEAD TO PRINTING IN QUICKDRAW GX

THE PALETTE MANAGER WAY

KON & BAL'S PUZZLE PAGE

MACINTOSH Q & A

NEW QUICKTIME COLUMN: TOP 10 QUICKTIME TIPS

Issue 13 March 1993

EDITORIAL STAFF

Editor-in-Cheek *Caroline Rose*

Technical Buckstopper *Dave Johnson*

Our Boss *Greg Joswiak*

His Boss *David Krathwohl*

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

Managing Editor *Monica Meffert*

Assistant Managing Editor *Cynthia Jasper*

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

Indexer *Ira Kleinberg*

ART & PRODUCTION

Production Manager *Hartley Lesser*

Art Director *Diane Wilcox*

Technical Illustration *Dave Olmos, John Ryan*

Formatting *Forbes Mill Press*

Printing *Wolfer Printing Company, Inc.*

Film Preparation *Aptos Post, Inc.*

Production *PrePress Assembly*

Photography *Sharon Beals, Lisa Jongewaard, Tom Sandborn*

Online Production *Cassi Carpenter*

Mark Jenkins of Rucker Huggins created this beautiful image depicting asynchronous routines by synchronizing four graphics applications: Adobe Photoshop, Adobe Illustrator, Ray Dream Designer, and Fractal Design Painter.

develop, The Apple Technical Journal, is a quarterly publication of Apple Computer's Developer Support Information group.

The *Developer CD Series* disc for March 1993 or later contains this issue and all back issues of *develop* along with the code that the articles describe. The *develop* issues and code are also available on AppleLink and via anonymous ftp on ftp.apple.com.

EDITORIAL Documentation matters. 2

LETTERS Apple DocViewer queries. What's on *your* mind? 4

ARTICLES **Asynchronous Routines on the Macintosh by Jim Luther** How to avoid the pitfalls of calling routines asynchronously. 5

Inside QuickTime and Component-Based Managers by Bill Guschwan Useful debugging and tracing techniques for QuickTime and the Component Manager. 34

Macintosh Debugging: The Belly of the Beast Revisited by Fred Huxham and Greg Marriott A supplement to the Belly of the Beast debugging article in Issue 8: four new tools explained. 54

Adventures in Color Printing by Dave Hersey A general strategy for printing color images that ensures the best possible quality. 64

DeviceLoop Meets the Interface Designer by John Powers This little-known System 7 routine can help you deal with multiple screen environments. 97

COLUMNS **Somewhere in QuickTime: Top 10 QuickTime Tips by John Wang** The first installment of a new column on QuickTime: hot tips from the masters. 31

Print Hints: Looking Ahead to QuickDraw GX by Pete ("Luke") Alexander Some things you should be aware of involving QuickDraw GX and its effect on printing. 52

The Veteran Neophyte: Tower of Babble by Dave Johnson Programming languages are just like natural languages, only different. 61

Graphical Truffles: The Palette Manager Way by Edgar Lee and Forrest Tanaka The Palette Manager need not be a mystery. Here's the scoop. 91

KON & BAL's Puzzle Page: Booting Blues by Konstantin Othmer and Bruce Leak Yet another elusive crasher bug that you'll never guess in a million years. 118

Q & A **Macintosh Q & A** Apple's Developer Support Center answers your product development questions. 104

INDEX 122

CAROLINE ROSE

Dear Readers,

In May of last year, I schmoozed with a lot of developers at Apple's Worldwide Developer's Conference, and one of the subjects that came up was documentation. I expressed my ideas on this subject somewhat hesitantly, because I thought the truths I was spouting were all pretty obvious, but I was surprised to find that several developers seemed enlightened by them and even suggested this as a topic for a *develop* editorial. So here goes.

But first, some motivation. If you're one of those developers who think no one reads manuals anyway, has it occurred to you that this might be a self-fulfilling prophecy? If manuals were better, maybe people would read them. Also, customers who *do* never read the manual will never learn the full power of your product (probably not *every* feature is self-explanatory) and will be that much quicker to move on when someone shows them the great things a rival product can do. More likely, people glance at the manual to get started and then thumb through it later when they want to explore certain features.

Also keep in mind that a shoddy manual will be seen as a reflection of the product as a whole: "If this is the best they could do on the manual, how good can their software be?" Don't fool yourself that only writers or editors will criticize a poorly done manual; any reader who has trouble learning from it will complain, and not just to themselves. While there are times when consistency may be the hobgoblin of small minds, it's often the case that inconsistent presentation or terminology will confuse readers and have them throwing your manual down in disgust and thinking your product is more complicated than it really is. And people who do know things like the difference between "its" and "it's" will wonder how well you debugged your code if you couldn't find mistakes like this in your manual. Basically, you won't look like a class act.

I'll state the following points with user documentation in mind, though most of them also apply to technical documentation for developers. Some points may be useful only to small companies, but there should be something here for everyone.

- Get a technical writer to write your documentation. Don't do it yourself — and try to talk the CEO or VP of Marketing out of doing it. Contrary to many people's opinions, writing a manual is

CAROLINE ROSE (AppleLink CROSE) has been writing and editing software documentation since many of you were rug rats. She began at a timesharing company, joined Apple in 1982 to write *Inside Macintosh*, and helped get NeXT off the ground in 1986. Back at Apple now, she has seven issues of *develop* under her belt and is still having a wonderful time. A transplanted New Yorker, Caroline visited the East Coast last October in time to see the leaves turn colors. She enjoyed doing the theater and museum thing in Manhattan and hitting some incredible restaurants and nightclubs — not to mention a deli whose smoked mozarella is the best this side of Naples. But the highlight was her visit to a friend's farm in Connecticut (sheep feeding beats sheet feeding any day!). Walking to an apple orchard and tasting fresh sweet cider was sheer rural bliss.•

not something any smart person can do; it's a skill like any other. Most likely you are no more qualified to write the documentation than a writer is qualified to write your code.

- Look over a relevant writing sample from your prospective writer. Awards, certificates, and years of experience go only so far: nothing will tell you whether you'll get a good manual as much as looking at past work. Ask how the material for the sample was gathered, who else contributed to it, and how heavily it was edited.
- Get the writer started early in the process — long before the feature set is frozen. Writers provide a valuable perspective of your product, not unlike that of product management. They'll help with the design of the product, telling you what features don't fit in with other ones and pointing out loopholes, inconsistencies, and other Bad Things. And they're typically excellent bug finders.
- Have the documentation edited by an editor. Unless they also happen to be editors, writers need their work checked like anyone else — and an electronic spelling or grammar check (while a good start) isn't enough.
- Test the result on users, after your product ships if not sooner (you can revise the documentation for the next printing). And don't be defensive: if only one out of ten "testers" turns up a particular problem, it may mean that 10% of your user base will have the same problem. Judge no misunderstanding as stupid; they're all valid, no matter how much you may disagree with them.

I could go on forever, but that's enough for now. Please make my day and let me know if you got anything of value out of this. Or criticize it if you like; I can use the practice in not being defensive.

Caroline Rose
Editor

SUBSCRIPTION INFORMATION

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

BACK ISSUES

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

LETTERS

TECH NOTES: WORD IS OUT

Why are the Tech Notes in Microsoft Word documents? Are you assuming all developers have Microsoft Word? I don't think this is a good assumption. Developers who don't have Microsoft Word would be required to either purchase it or get an illegal copy. Or I suppose they might be able to use their favorite word processor and convert the Tech Notes if such converters exist.

Big developers may have the capital to purchase Microsoft Word but small or starting developers may not, especially those enthusiastic and creative programmers in school. It would be a shame to force them to get an illegal copy of Microsoft Word so that they could learn the same wonderful magic tricks that others get from the Tech Notes.

Is it possible to produce the Tech Notes in a minimal text editor such as TeachText or DocMaker? Or better yet, why not use Apple DocViewer like the *New Inside Macintosh* documents?

— Hoon Im

This is a timely question, as the format of Tech Notes on the CD has changed. But first, some background.

There are several reasons why we distributed Tech Notes as Microsoft Word documents. Internally we use Microsoft Word as the authoring tool for Tech Notes because of its relatively powerful formatting abilities and ease of use. It also turns out that most word processing packages, such as MacWrite® II, have translators that do a reasonable job on Microsoft Word documents, so most people have access to the information. We're firmly against *pirating software!*

Also, our primary commitment has been to providing the highest quality technical material possible; rather than focusing on format conversion, we chose to improve the overall content and organization of the Tech Notes. Only then were we ready to turn our full attention to the question of format.

You mention Apple DocViewer as a possible alternative format — we have in fact converted the Tech Notes into Apple DocViewer format (take a look on the CD). Over time this will be improved to provide better indexing and cross-reference facilities — whose absence we've been painfully aware of in the Microsoft Word format.

— Neil Day, Tech Note Pooh-Bah

LICENSING DOCVIEWER

We are a long-time Macintosh educational software developer. We've traditionally converted our printed documentation to HyperCard® for on-line use by our customers. I wondered to whom we should speak to request developer licensing of Apple DocViewer?

—Rhett Tindall

Apple DocViewer documents are sourced from several word processors. These documents must be processed in another application before they become DocViewer documents. This application is currently not of commercial quality and is for Apple internal use only. However, we're in the process of investigating whether to refine the application and make it available outside Apple. This process may take some time and may not result in providing the software to external parties. Please stay tuned!

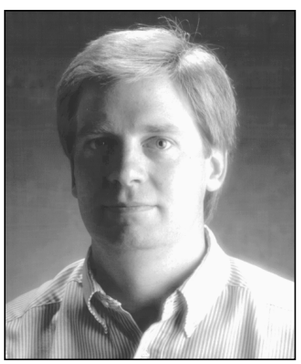
— In-Yung Kim

GIVE US A PIECE OF *YOUR* MIND

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

ASYNCHRONOUS ROUTINES ON THE MACINTOSH

The Macintosh has always supported asynchronous calls to many parts of its operating system. This article expands on the information found in Inside Macintosh by telling when, why, and how you should use functions asynchronously on the Macintosh. It includes debugging hints and solutions to problems commonly encountered when asynchronous calls are used.



JIM LUTHER

When calling a routine synchronously, your program passes control to the routine and doesn't continue execution until the routine's work has completed (either successfully or unsuccessfully). This would be like giving someone a task and then watching them perform that task. Although the task is eventually completed, you don't get anything done while you watch.

On the other hand, when calling a routine asynchronously, your program passes control to the routine, and the program's request is placed in a queue or, if the queue is empty, executed immediately; in either case, control returns to the program very quickly, even if the request can't be executed until later. The system processes any queued requests while your program is free to continue execution, then interrupts you later when the request is completed. This is like giving someone a task and going back to your work while they finish the task. In most cases, it results in more work being accomplished during the same period of time. Figure 1 illustrates the difference between synchronous and asynchronous calls.

One situation in which you shouldn't use synchronous calls is when you don't know how long it may take for the operation to complete, as with the PPC Toolbox's PPCInform function, for example. PPCInform won't complete until another program attempts to start a session with your program. This could happen immediately, but the chances are far greater that it won't. If PPCInform is called synchronously, it appears that the system has locked up because the user won't get control back until the call completes. If you call PPCInform asynchronously, it doesn't matter if the function doesn't complete for minutes, hours, or even days — your program (and the rest of the system) can continue normally.

JIM LUTHER works in Apple Developer Technical Support, focusing on AppleTalk, the File Manager, and other lower regions of the operating system. Jim uses a heap/stack-based organizational model in his office: there are heaps or stacks of books, papers, disks, and hardware on every square inch of shelf space and over most of the floor. He was last seen muttering to himself "Now where did I put that . . .?"•

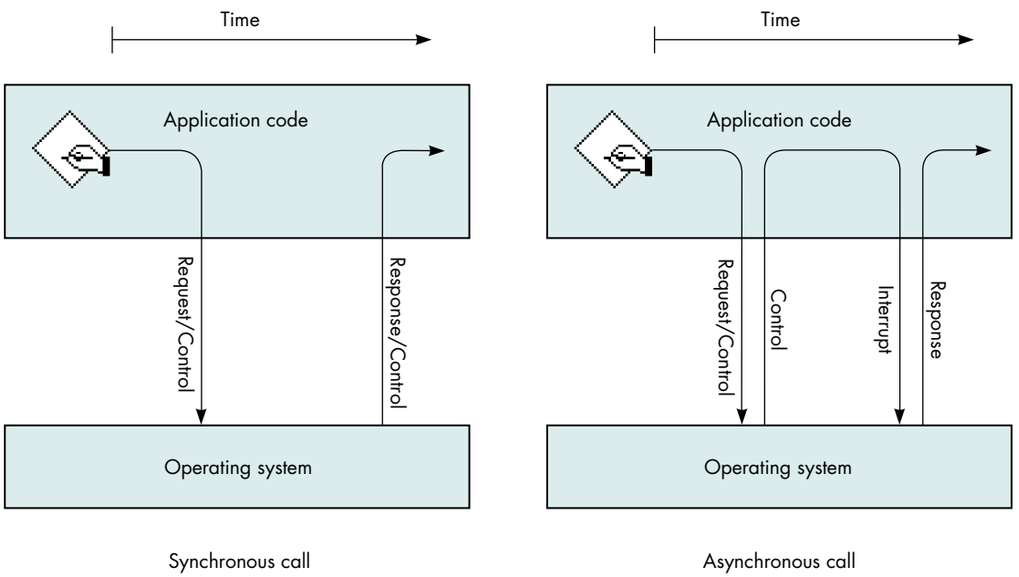


Figure 1
How Synchronous and Asynchronous Calls Work

You should also avoid synchronous calls when you can't know the state of the service you've asked for. Program code that's part of a completion routine, VBL task, Time Manager task, Deferred Task Manager task, or interrupt handler is executed at what's commonly called *interrupt time*. Synchronous calls made at interrupt time often result in deadlock. (See "Deadlock.") An asynchronous call can solve the problem: if the service you call is busy handling another request, your asynchronous request is queued and your program code can give up control (that is, the completion routine or task your code is part of can end), letting the service complete the current request and eventually process your request.

Routines called synchronously are allowed to move memory, while routines called asynchronously purposely avoid moving memory so that they can be called at interrupt time. For example, the File Manager's PBHOpen routine may move memory when called synchronously, but won't when called asynchronously. If your code is executing in an environment where memory can't be moved (for example, at interrupt time), you must call routines asynchronously to ensure that they don't move memory.

At this time, the various lists in *Inside Macintosh* of "Routines That May Move or Purge Memory," "Routines and Their Memory Behavior," and "Routines That

DEADLOCK
BY GORDON SHERIDAN

Deadlock is a state in which each of two or more processes is waiting for one of the other processes to release some resource necessary for its completion. The resource may be a file, a global variable, or even the CPU. The process could, for example, be an application's main event loop or a Time Manager task.

When deadlock occurs on the Macintosh, usually at least one of the processes is executing as the result of an interrupt. VBL tasks, Time Manager tasks, Deferred Task Manager tasks, completion routines, and interrupt handlers can all interrupt an application's main thread of execution. When the interrupted process is using a resource that the interrupting process needs, the processes are deadlocked.

For example, suppose a Time Manager task periodically writes data to a file by making a synchronous Write request, and an application reads the data from its main event loop. Depending on the frequency of the task and the activity level of the File Manager, the Time Manager task may often write successfully. Inevitably, however, the Time Manager task will interrupt the application's Read request and deadlock will occur.

Because the File Manager processes only one request at a time, any subsequent requests must wait for the current request to complete. In this case, the synchronous request made by the Time Manager task must wait for the application's Read request to complete before its Write request will be processed. Unfortunately, the File Manager must wait for the Time Manager task to complete before it can resume execution. Each process is now waiting for the other to complete, and they'll continue to wait forever.

Synchronous requests at interrupt time tend to produce deadlock, because the call is queued for processing and then the CPU sits and spins, waiting for an interrupt to occur, which signals that the request has been completed. If interrupts are turned off, or if a previous pending request can't finish because it's waiting to resume execution after the interrupt, the CPU will wait patiently (and eternally) for the request to finish — until you yank the power cord from the wall.

Should Not Be Called From Within an Interrupt" are either incomplete or incorrect and can't be trusted entirely. The reasons why a system routine can't be called at interrupt time include: the routine may move memory; the routine may cause a deadlock condition; the routine is not reentrant. This article shows how to postpone most system calls until a safe time. You're encouraged to call as few system routines at interrupt time as possible.

The routines discussed in this article are low-level calls to the File Manager, the Device Manager (including AppleTalk driver, Serial Driver, and disk driver calls), and the PPC Toolbox. All these routines take the following form:

```
FUNCTION SomeFunction (pbPtr: aParamBlockPtr; async: BOOLEAN): OSErr;
```

Routines of this form are executed synchronously when async = FALSE or asynchronously when async = TRUE.

DETERMINING ASYNCHRONOUS CALL COMPLETION

Your program can use two methods to determine when an asynchronous call has completed: periodically poll for completion (check the ioResult field of the parameter block passed to the function) or use a completion routine. Both methods enable your program to continue with other operations while waiting for an asynchronous call to complete.

POLLING FOR COMPLETION

Polling for completion is a simple method to use when you have only one or two asynchronous calls outstanding at a time. It's like giving someone a task and calling them periodically to see if they've completed it. When your program fills in the parameter block to pass to the function, it sets the ioCompletion field to NIL, indicating that there's no completion routine. Then, after calling the function asynchronously, your program only needs to poll the value of the ioResult field of the parameter block passed to the function and wait for it to change:

- A positive value indicates the call is either still queued or in the process of executing.
- A value less than or equal to 0 (noErr) indicates the call has completed (either with or without an error condition).

Polling is usually straightforward and simple to implement, which makes the code used to implement polling easy to debug. The following code shows an asynchronous PPCInform call and how to poll for its completion:

```
PROCEDURE MyPPCInform;
   VAR
      err: OSErr; { Error conditions are ignored in this procedure }
                  { because they are caught in PollForCompletion. }

BEGIN
   gPPCParamBlock.informParam.ioCompletion := NIL;
   gPPCParamBlock.informParam.portRefNum := gPortRefNum;
   gPPCParamBlock.informParam.autoAccept := TRUE;
   gPPCParamBlock.informParam.portName := @gPPCPort;
   gPPCParamBlock.informParam.locationName := @gLocationName;
   gPPCParamBlock.informParam.userName := @gUserName;
   err := PPCInform(PPCInformPBPtr(@gPPCParamBlock), TRUE);
END;
```

In this code, MyPPCInform calls the PPCInform function asynchronously with no completion routine (ioCompletion is NIL). The program can then continue to do other things while periodically calling the PollForCompletion procedure to find out when the asynchronous call completes.

```
PROCEDURE PollForCompletion;
BEGIN
   IF gPPCParamBlock.informParam.ioResult <= noErr THEN
      BEGIN    { The call has completed. }
         IF gPPCParamBlock.informParam.ioResult = noErr THEN
            BEGIN
               { The call completed successfully. }
            END
         ELSE
            BEGIN
               { The call failed, handle the error. }
            END;
      END;
END;
```

PollForCompletion checks the value of the ioResult field to find out whether PPCInform has completed. If the call has completed, PollForCompletion checks for an arror condition and then performs an appropriate action.

There are three important things to note in this example of polling for completion:

- The parameter block passed to PPCInform, gPPCParamBlock, is a program global variable. Since the parameter block passed to an asynchronous call is owned by the system until the call completes, the parameter block must not be declared as a local variable within the routine that makes the asynchronous call. The memory used by local variables is released to the stack when a routine ends, and if that part of the stack gets reused, the parameter block, which could still be part of an operating system queue, can get trashed, causing either unexpected results or a system crash. Always declare parameter blocks globally or as nonrelocatable objects in the heap.
- Calls to PollForCompletion must be made from a program loop that's not executed completely at interrupt time. This prevents deadlock. You don't necessarily have to poll from an application's event loop (which is executed at noninterrupt time), but if you poll from code that executes at interrupt time, that code must give up control between polls.
- PollForCompletion checks the ioResult field of the parameter block to determine whether PPCInform completed and, if it completed, to see if it completed successfully.

One drawback to polling for completion is latency. When the asynchronous routine completes its job, your program won't know it until the next time you poll. This can be wasted time. For example, assume you give someone a task and ask them if they're done (poll) only once a day: if they finish the task after an hour, you won't find out

FUNCTION RESULTS AND FUNCTION COMPLETION

BY SCOTT BOYD AND JIM LUTHER

Not all function results are equal. Ignore some, pay attention to others. Ignore function results from asynchronous File Manager and PPC Toolbox calls. They contain no useful information. To get useful result information, wait for the call to complete, then check ioResult or register D0; both contain the result.

Both the File Manager and the PPC Toolbox will always call your completion routine if you specified one. If you didn't supply one, and instead are polling, test ioResult in your parameter block. The call has completed if ioResult is less than or equal to noErr.

Don't ignore function results from asynchronous Device Manager calls (for example, AppleTalk driver, Serial Driver, and disk driver calls). The function result tells you whether the Device Manager successfully delivered your request to the device driver. Success is indicated by noErr; any other value indicates failure.

The system calls your completion routine only if the Device Manager successfully delivered your request to the driver. On completion, check whether your call succeeded by looking in ioResult or register D0.

they've completed the task until 23 hours later (a 23-hour latency). To avoid latency, use completion routines instead of polling ioResult to find out when a routine completes.

USING COMPLETION ROUTINES

Making an asynchronous call with a completion routine is only slightly more complex than polling for completion. A completion routine is a procedure that's called as soon as the asynchronous function completes its task. When your program fills in the parameter block to pass to the function, it sets the ioCompletion field to point to the completion routine. Then, after calling the function asynchronously, your program can continue. When the function completes, the system interrupts the program that's running and the completion routine is executed. (There are some special things you need to know about function results to use this model; see "Function Results and Function Completion.")

Since the completion routine is executed as soon as the function's task is complete, your program finds out about completion immediately and can start processing the results of the function. Using a completion routine is like giving someone a task and then asking them to call you as soon as they've completed it.

Because a completion routine may be called at interrupt time, it can't assume things that most application code can. When a completion routine for an asynchronous function gets control, the system is in the following state:

- On entry, register A0 points to the parameter block used to make the asynchronous call.

- Your program again owns the parameter block used to make the asynchronous call, which means you can reuse the parameter block to make another asynchronous call (see the section "Call Chaining" later in this article).
- Both register D0 and ioResult in the parameter block contain the result status from the function call.
- For completion routines called by the File Manager or Device Manager, the A5 world is undefined and must be restored before the completion routine uses any application global variables.

Since completion routines execute at interrupt time, they must follow these rules:

- They must preserve all registers except A0, A1, and D0-D2.
- They can't call routines that can directly or indirectly move memory, and they can't depend on the validity of handles to unlocked blocks.
- They shouldn't perform time-consuming tasks, because interrupts may be disabled. As pointed out in the Macintosh Technical Note "NuBus™ Interrupt Latency (I Was a Teenage DMA Junkie)," disabling interrupts and taking over the machine for long periods of time "almost always results in a sluggish user interface, something which is not usually well received by the user." Some ways to defer time-consuming tasks are shown later in this article.
- They can't make synchronous calls to device drivers, the File Manager, or the PPC Toolbox for the reasons given earlier.

PPC Toolbox completion routines. The PPC Toolbox simplifies the job of writing completion routines. When a PPC Toolbox function is called asynchronously, the current value of register A5 is stored. When the completion routine for that call is executed by the PPC Toolbox, the stored A5 value is restored and the parameter block pointer used to make the call is passed as the input parameter to the completion routine.

A completion routine called by the PPC Toolbox has this format in Pascal:

```
PROCEDURE MyCompletionRoutine (pbPtr: PPCParamBlockPtr);
```

PPC Toolbox completion routines are still called at interrupt time and so must follow the rules of execution at interrupt time.

The following code shows an asynchronous PPCInform call and its completion routine.

```
PROCEDURE InformComplete (pbPtr: PPCParamBlockPtr);
BEGIN
   IF pbPtr^.informParam.ioResult = noErr THEN
      BEGIN
         { The PPCInform call completed successfully. }
      END
   ELSE
      BEGIN
         { The PPCInform call failed; handle the error. }
      END;
END;

PROCEDURE DoPPCInform;
   VAR
      err: OSErr; { Error conditions are ignored in this procedure }
                  { because they are caught in InformComplete. }

BEGIN
   gPPCParamBlock.informParam.ioCompletion := @InformComplete;
   gPPCParamBlock.informParam.portRefNum := gPortRefNum;
   gPPCParamBlock.informParam.autoAccept := TRUE;
   gPPCParamBlock.informParam.portName := @gPPCPort;
   gPPCParamBlock.informParam.locationName := @gLocationName;
   gPPCParamBlock.informParam.userName := @gUserName;
   err := PPCInform(PPCInformPBPtr(@gPPCParamBlock), TRUE);
END;
```

In this code, DoPPCInform calls PPCInform asynchronously with a completion routine (ioCompletion contains a pointer to InformComplete). The program can then continue to do other things.

When PPCInform completes, control is passed to InformComplete with a pointer to gPPCParamBlock. InformComplete checks the result returned by PPCInform and then performs an appropriate action.

Here are the important things to note in this example of a PPC Toolbox completion routine:

- The parameter block gPPCParamBlock is declared globally for the reasons given earlier in the section "Polling for Completion."
- InformComplete checks the ioResult field of the parameter block to determine whether PPCInform completed successfully.

File Manager and Device Manager completion routines in high-level languages. File Manager and Device Manager completion routines written in a

COMPLETION ROUTINE ADDRESS GENERATION When you fill a parameter block's ioCompletion field with the address of a completion routine, your compiler has to calculate the address of the completion routine. Most compilers generate that address either as a PC-relative reference (the address of the routine's entry point within the local code segment) or as an A5-relative reference (the address of the routine's jump table entry). If your compiler generates an A5-relative reference, the code that generates the address of the completion routine *must* run with the program's A5 world set up.

MPW Pascal defaults to using PC-relative references when routines are in the same segment and uses A5-relative references when routines are in a different segment. MPW C defaults to using

high-level language such as Pascal or C are more complicated than PPC Toolbox completion routines. They must take additional steps to get the value in register A0 and, if program global variables will be used, restore register A5 to the application's A5 value. The reason for this is that File Manager and Device Manager completion routines are called with the pointer to the call's parameter block in register A0 and with the A5 world undefined.

In most high-level languages, registers A0, A1, and D0-D2 are considered scratch registers by the compiler and aren't preserved across routine calls. For this reason, you should not depend on register values as input parameters to routines written in a high-level language. Examples of completion routines in *Inside Macintosh* and in several Macintosh Technical Notes use short inline assembly routines to retrieve the value of register A0, in the following manner:

```
FUNCTION GetPBPtr: ParmBlkPtr;
{ Return the pointer value in register A0. }
INLINE $2E88;  { MOVE.L A0,(A7) }

PROCEDURE MyCompletionRoutine;
{ This procedure gets called when an asynchronous call completes. }
   VAR
      pbPtr: ParmBlkPtr;

BEGIN
   pbPtr := GetPBPtr;   { Retrieve the value in register A0. }
   DoWork(pbPtr);       { Call another routine to do the actual work. }
END;
```

Although the GetPBPtr inline assembly routine works with today's compilers, be careful, because register A0 could be used by the compiler for some other purpose before the statement with the inline assembly code is executed. As shown in the previous example, you can minimize the chances of the compiler using a register before you retrieve its value by retrieving the register value in the completion routine's first statement and then doing as little as possible within the completion routine (call another routine to do any additional work).

The safest way to use register values as input parameters to completion routines written in a high-level language is to use a completion routine written in assembly language that calls a routine written in a high-level language. The following record type allows File Manager and Device Manager completion routines to be written in high-level languages such as C or Pascal with only one small assembly language routine. This record also holds the application's A5 value so that the completion routine can restore A5 and application globals can be accessed from within the completion routine.

A5-relative references. THINK Pascal and THINK C always use A5-relative references. MPW Pascal and MPW C allow you to change their default method with the **-b** compiler option.•

```
TYPE
   extendedPBPtr = ^extendedPB;
   extendedPB = RECORD
         ourA5:         LONGINT;    { Application's A5 }
         ourCompletion: ProcPtr;    { Address of the completion routine }
                                    { written in a high-level language }
         pb:         ParamBlockRec; { Parameter block used to make call }
      END;
```

PreCompletion, a small assembly language routine, is used as the completion routine for all File Manager and Device Manager asynchronous calls (PreCompletion comes preassembled and ready to link with your C or Pascal code on the *Developer CD Series* disc). PreCompletion preserves the A5 register, sets A5 to the application's A5, calls the designated Pascal completion routine with a pointer to the parameter block used to make the asynchronous call, and then restores the A5 register:

```
PreCompletion  PROC  EXPORT
      LINK     A6,#0             ; Link for the debugger.
      MOVEM.L  A5,-(SP)          ; Preserve A5 register.
      MOVE.L   A0,-(SP)          ; Pass PB pointer as the parameter.
      MOVE.L   -8(A0),A5         ; Set A5 to passed value (ourA5).
      MOVE.L   -4(A0),A0         ; A0 = real completion routine address.
      JSR      (A0)              ; Transfer control to ourCompletion.
      MOVEM.L  (SP)+,A5          ; Restore A5 register.
      UNLK     A6                ; Unlink.
      RTS                        ; Return.
      STRING   ASIS
      DC.B     $8D,'PreCompletion'  ; The debugger string.
      DC.W     $0000
      STRING   PASCAL
      ENDP
      END
```

Before an application makes an asynchronous call, it initializes the extendedPB record with the application's A5 and the address of the high-level language's completion routine. The ioCompletion field of the extendedPB record's parameter block is initialized with the address of PreCompletion:

```
myExtPB.ourA5 := SetCurrentA5;
myExtPB.ourCompletion := @MyCompletionRoutine;
myExtPB.pb.ioCompletion := @PreCompletion;
```

The high-level language's completion routine called by PreCompletion has this format in Pascal:

```
PROCEDURE MyCompletionRoutine (pbPtr: ParmBlkPtr);
```

When MyCompletionRoutine is called, register A5 has been set to the stored application A5 and pbPtr points to the parameter block (within the extended parameter block) used to make the asynchronous call.

The rest of this article shows how to use asynchronous calls and completion routines to your program's advantage and describes various techniques for working around the limitations imposed on completion routines.

THE BIG THREE TECHNIQUES

There are lots of techniques you can use when working with asynchronous calls. Most are useful for solving only one or two programming problems. This section describes the three most useful techniques — the use of operating system queues, call chaining, and extended parameter blocks.

OPERATING SYSTEM QUEUES

After reading the description of operating system queues in *Inside Macintosh* Volume II, you might assume they're for use only by the operating system. Wrong! Any program can create an OS queue for its own purposes. OS queues are very useful in interrupt-time code such as completion routines, because the two routines that manipulate OS queues, Enqueue and Dequeue, have the following characteristics:

- They disable all interrupts while they update the queue. This is very important because it prevents race conditions between interrupt and noninterrupt code accessing the queue. (See "Race Conditions and OS Queues.")
- They can be called at interrupt time, because they don't move memory — they only manipulate a linked list of queue elements.
- They're very fast and efficient, so they won't be time-consuming operations in your completion routines.

An OS queue owned by your program can hold queue elements defined by the system or queue elements of your own design. A queue element is a record that starts with two fields, qLink and qType. The qLink field is a QElemPtr that links queue elements together while they're in an OS queue. The qType field is an integer value that identifies the queue element type. In OS queues owned by your program, you may not need to use the qType field unless the OS queue can hold more than one type of queue element. Here's how the system defines a queue element:

```
QElem = RECORD
      qLink:   QElemPtr;   { Link to next queue element. }
      qType:   INTEGER;    { Queue element type. }
      { Add your data fields here. }
   END;
```

RACE CONDITIONS AND OS QUEUES

When two or more processes share the same data, you must be careful to avoid *race conditions*. A race condition exists when data is simultaneously accessed by two processes. On the Macintosh, the two processes are typically program code running with interrupts enabled and code executing at interrupt time (such as a completion routine).

To prevent race conditions, you must have a method of determining ownership of the shared data. A shared global flag isn't safe because there can be a race condition with the flag. For example, the following code can't be used to claim ownership of a record:

```
{ This can cause a race condition. }
IF gRecFree THEN { Is record in use? }
   BEGIN { It wasn't when we checked. }
      gRecFree := FALSE;   { Claim record. }
      { Use record. }
      gRecFree := TRUE;    { Release record. }
   END;
```

A race condition can occur in this code because there's a small window of time between when the IF statement's expression is evaluated and when the record is claimed. During this time the program can be interrupted. The only way to prevent race conditions is to make the process of checking for and claiming ownership a step that can't be interrupted.

OS queues and the Operating System Utility routines Enqueue and Dequeue provide a safe way to claim ownership of data. Enqueue and Dequeue briefly disable interrupts while manipulating the queue, so they're safe from race conditions.

To use an OS queue to protect data from race conditions, make the data part of a queue element and put the queue element into an OS queue. Whenever any part of the program wants to manipulate data in the queue element, it attempts to remove the queue element from the OS queue. If the queue element isn't in the queue and so can't be removed, that means another process currently has ownership of the queue element and the data within it.

The following record types are some of the system-defined queue elements: ParamBlockRec, CInfoPBRec, DTPBRec, HParamBlockRec, FCBPBRec, WDPBRec, CMovePBRec, MPPParamBlock, ATPParamBlock, XPPParamBlock, DSPParamBlock, EParamBlock, PPCParamBlockRec, TMTask, DeferredTask, and VBLTask.

To use an OS queue in your program, you need to allocate a queue header (QHdr) variable and possibly define your own queue element type:

```
TYPE
   { Define a queue element type. }
   MyQElemRecPtr = ^MyQElemRec;
   MyQElemRec = RECORD
         qLink:   QElemPtr;
         qType:   INTEGER;
         myData:  myDataType; { Put any data fields you want here. }
      END;
VAR
```

```
   { Allocate a queue element and a queue header. }
   myQElem:       MyQElemRec;
   myOSQueueHdr:  QHdr;
```

You must initialize the queue header before it's used by setting its qHead and qTail fields to NIL:

```
{ Initialize the OS queue. }
myOSQueueHdr.qHead := NIL;
myOSQueueHdr.qTail := NIL;
```

The queue element can then be added to the OS queue:

```
{ Add myQElem to the queue. }
Enqueue(QElemPtr(@myQElem), @myOSQueueHdr);
```

This code shows how to remove a queue element (in this example, the first item in the queue) from an OS queue before using it:

```
VAR
   myQElemPtr: MyQElemRecPtr;

myQElemPtr := MyQElemRecPtr(myOSQueueHdr.qHead);
IF myQElemPtr <> NIL THEN  { Make sure we have a queue element. }
   BEGIN
      IF Dequeue(QElemPtr(myQElemPtr), @myOSQueueHdr) = noErr THEN
         BEGIN
            { We successfully removed the queue element from the queue, }
            { so we can use myQElemPtr^.myData. }
            { In this example, we'll put the queue element back in }
            { the queue when we're done with it. }
            Enqueue(QElemPtr(myQElemPtr), @myOSQueueHdr);
         END
      ELSE
         BEGIN
            { Someone else just claimed the queue element between the }
            { two IF statements and we just avoided a race condition! }
            { Try again later. }
         END;
   END;
```

OS queues owned by your program can be used for many purposes, including these:

- Completion routines can schedule work to be done by your application's event loop by putting requests into an OS queue.

- Extra buffers or parameter blocks needed by completion routines can be put into an OS queue by code called from the program's event loop. These buffers or parameter blocks can be safely claimed and used by code running at interrupt time.
- Completion routines can schedule the processing of a completed call by putting the parameter block used to make the call into an OS queue. This is useful when the processing might move memory or take too much time and so can't be performed in the completion routine.
- Data accessed and manipulated by both interrupt code and noninterrupt code can be protected from race conditions if it's stored in a queue element and the Dequeue and Enqueue routines are used to claim and release ownership of the data from an OS queue (as described earlier in "Race Conditions and OS Queues").

CALL CHAINING

When a multistep operation is performed via multiple asynchronous calls with completion routines, it's called *call chaining*. Each asynchronous call's completion routine reuses the parameter block passed to the completion routine to make the next asynchronous call. Call chaining from completion routines allows your program to start the next step in a multistep operation with no latency (see Figure 2).

To use call chaining, you must design your call chain; that is, you must decide the order of the asynchronous calls you want to make. For each completion routine, determine what step should be taken if the previous call completed successfully with no error condition and what step should be taken if the previous call completed with an error.

A chained call sequence may have several end points or breaks in the chain, depending on what you're trying to accomplish and what conditions are encountered along the way. For example, you may not want to make another asynchronous call because an error condition occurred, because the next step your program needs to take involves a call that can't be made at interrupt time, or because all steps were completed successfully. The easiest way for your chained call sequence to pass control back to noninterrupt code is through an OS queue. This technique is shown in the section "Putting the Big Three Together."

EXTENDING PARAMETER BLOCKS

Unless you do a little extra work, a completion routine is somewhat isolated from the rest of your program. The only data accessible to a completion routine when it executes is the parameter block used to make the asynchronous call and, if you preserve and restore A5, the program's global variables. As noted before, you must be careful to avoid race conditions when accessing global variables.

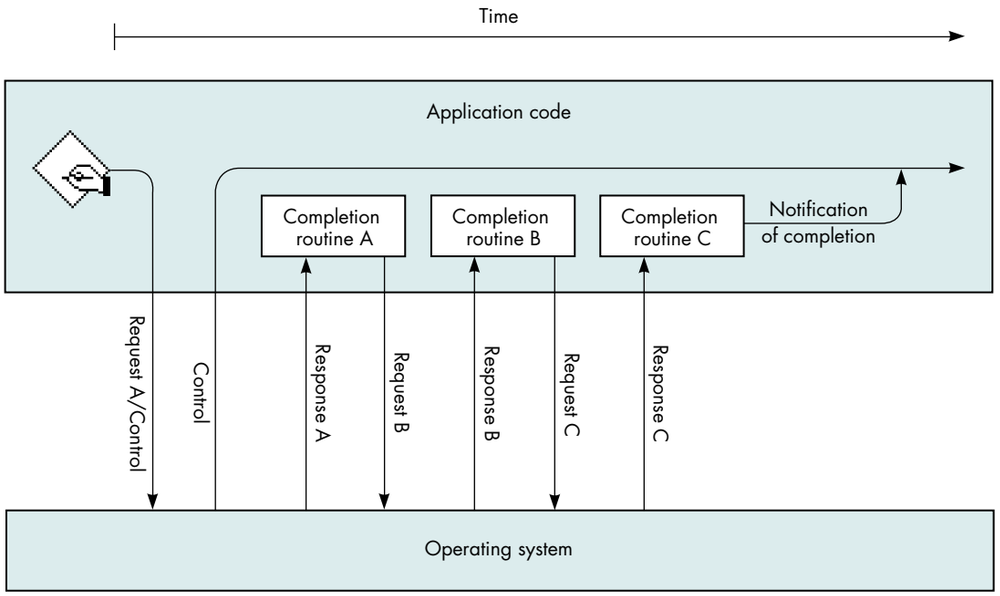


Figure 2
Call Chaining

You can extend a parameter block by attaching your own data to the end of a parameter block, like this:

```
TYPE
   myPBRecPtr = ^myPBRec;
   myPBRec = RECORD
         pb:      ParamBlockRec;
         myData:  myDataType; { Put any data type you want here. }
      END;
```

From within a completion routine, using the extended fields is easy:

```
IF thePBRecPtr^.pb.ioResult = noErr THEN
   thePBRecPtr^.myData := kSomeValue;
```

Extending a parameter block has several benefits for asynchronous program code:

- By extending a parameter block to include all variables used by the routine, you can reduce the amount of stack space used by completion routines.

- By keeping all data associated with a particular session in the extended parameter block, you can support multiple independent sessions.
- By putting values needed by a completion routine in an extended parameter block instead of in program global variables, you can prevent race conditions. This provides noninterrupt code and interrupt code with a safe method to communicate.

PUTTING THE BIG THREE TOGETHER

Now that you know about OS queues, call chaining, and extending parameter blocks, let's look at a simple example of how these techniques can be used together. PPC Toolbox calls, being slightly simpler, are used in this example.

In the example, the program is to receive and accept a PPC session request, read some data, process the data, and then close the connection. To accomplish this, the program calls PPCInform asynchronously with a completion routine, has PPCInform's completion routine call PPCRead asynchronously with a completion routine, and then has PPCRead's completion routine schedule processing of the data by putting a request into an OS queue. After the data is removed from the queue and processed in the application's main event loop, the program calls PPCClose asynchronously with a completion routine and has PPCClose's completion routine call PPCInform again to wait for another connection.

We begin with an extended PPC parameter block record that can hold all the data the program needs to access from the various procedures:

```
CONST
   kPPCIOBufSize = 1024;   { Size of the I/O buffer. }
TYPE
   PPCIOBuffer = ARRAY[1..kPPCIOBufSize] OF SignedByte;
   PPCSessRecPtr = ^PPCSessRec;
   PPCSessRec = RECORD
         pb:               PPCParamBlockRec; { The pb must be first. }
         err:              OSErr;            { To catch results. }
         sessPortName:     PPCPortRec;       { Port name returned to }
                                             { PPCInform. }
         sessLocationName: LocationNameRec;  { Location name returned }
                                             { to PPCInform. }
         sessUserName:     Str32;            { User name returned to }
                                             { PPCInform. }
         buffer:           PPCIOBuffer;      { R/W buffer used by this }
                                             { session. }
      END;
```

Next, we declare the global variables used in this example:

```
VAR
   gQuitting:     BOOLEAN;       { True when no new sessions should }
                                 { be allowed. }
   gPortRefNum:   PPCPortRefNum; { PPC port reference number from }
                                 { PPCOpen. }
   gReadQueue:    QHdr;          { Where PPCRead parameter blocks are }
                                 { scheduled to be processed. }
   gDoneQueue:    QHdr;          { Where parameter blocks are put when }
                                 { completion routines are done with }
                                 { them. }
```

Several procedures are used in the example: DoPPCInform, InformComplete, ReadComplete, ProcessPPCData, EndComplete, and HandlePPCErrors. Not shown in this article is the program code for such operations as opening the PPC port, setting gQuitting to FALSE, and initializing the two OS queue headers before DoPPCInform is called.

DoPPCInform simply fills in the parameter block, previously allocated by the program and passed to DoPPCInform, and calls PPCInform asynchronously with InformComplete as the completion routine. Any errors returned by PPCInform will be handled by InformComplete.

```
PROCEDURE DoPPCInform (pbPtr: PPCSessRecPtr);
BEGIN
   { Call PPCInform. }
   PPCInformPBPtr(pbPtr)^.ioCompletion := @InformComplete;
   PPCInformPBPtr(pbPtr)^.portRefNum := gPortRefNum;
   PPCInformPBPtr(pbPtr)^.autoAccept := TRUE;
   PPCInformPBPtr(pbPtr)^.portName := @pbPtr^.sessPortName;
   PPCInformPBPtr(pbPtr)^.locationName := @pbPtr^.sessLocationName;
   PPCInformPBPtr(pbPtr)^.userName := @pbPtr^.sessUserName;
   { Error conditions are ignored in this procedure because they are }
   { caught in InformComplete. }
   pbPtr^.err := PPCInformAsync(PPCInformPBPtr(pbPtr));
   { Continued at InformComplete. }
END;
```

InformComplete is called when PPCInform completes. InformComplete first checks for errors from PPCInform. If the result is noErr, InformComplete fills in the parameter block and calls PPCRead asynchronously with ReadComplete as the completion routine. Any errors returned by PPCRead will be handled by ReadComplete. If PPCInform failed (the result is not noErr), InformComplete puts

the parameter block into gDoneQueue, where the error condition can be handled from the program's event loop.

```
PROCEDURE InformComplete (pbPtr: PPCSessRecPtr);
BEGIN
   IF PPCInformPBPtr(pbPtr)^.ioResult = noErr THEN
      { The PPCInform call completed successfully. }
      BEGIN
         { Call PPCRead. }
         PPCReadPBPtr(pbPtr)^.ioCompletion := @ReadComplete;
         { We're reusing the same parameter block, so the }
         { sessRefNum is already filled in. }
         PPCReadPBPtr(pbPtr)^.bufferLength := kPPCIOBufSize;
         PPCReadPBPtr(pbPtr)^.bufferPtr := @pbPtr^.buffer;
         { Error conditions are ignored in this procedure because they }
         { are caught in ReadComplete. }
         PPCSessRecPtr(pbPtr)^.err := PPCReadAsync(PPCReadPBPtr(pbPtr));
         { Continued at ReadComplete. }
      END
   ELSE
      { The PPCInform call failed. Drop the parameter block in the }
      { "done" queue for handling later. }
      Enqueue(QElemPtr(pbPtr), @gDoneQueue);
      { Dequeued by HandlePPCErrors. }
END;
```

ReadComplete is called when PPCRead completes. ReadComplete first checks for errors from PPCRead. If the result is noErr, ReadComplete puts the parameter block into gReadQueue. If PPCRead failed (the result is not noErr), ReadComplete puts the parameter block into gDoneQueue. In either case, the information queued is handled from the program's event loop.

```
PROCEDURE ReadComplete (pbPtr: PPCParamBlockPtr);
BEGIN
   IF PPCReadPBPtr(pbPtr)^.ioResult = noErr THEN
      { The PPCRead call completed successfully. Drop the }
      { parameter block in the "read" queue for handling later. }
      Enqueue(QElemPtr(pbPtr), @gReadQueue)
      { Dequeued by ProcessPPCData. }
   ELSE
      { The PPCRead call failed. Drop the parameter block in the }
      { "done" queue for handling later. }
      Enqueue(QElemPtr(pbPtr), @gDoneQueue)
      { Dequeued by HandlePPCErrors. }
END;
```

ProcessPPCData is called regularly from the program's event loop. If gReadQueue contains a parameter block, ProcessPPCData removes the parameter block from the queue and processes the data read in the PPCSessRec's buffer. After processing the data, ProcessPPCData calls PPCEnd asynchronously with EndComplete as the completion routine. Any errors returned by PPCEnd will be handled by EndComplete.

```
PROCEDURE ProcessPPCData;
   VAR
      pbPtr: PPCSessRecPtr;

BEGIN
   { Check for a parameter block in the queue. }
   IF gReadQueue.qHead <> NIL THEN
      BEGIN
         { Get the PPCSessRec and remove it from the queue. }
         pbPtr := PPCSessRecPtr(gReadQueue.qHead);
         IF Dequeue(QElemPtr(pbPtr), @gReadQueue) = noErr THEN
            BEGIN
               { Process PPCReadPBPtr(pbPtr)^.actualLength bytes of }
               { data in the data buffer, pbPtr^.buffer, here. }
               { Then call PPCEnd to end the session. }
               PPCEndPBPtr(pbPtr)^.ioCompletion := @EndComplete;
               { Error conditions are ignored in this procedure because }
               { they are caught in EndComplete. }
               pbPtr^.err := PPCEndAsync(PPCEndPBPtr(pbPtr));
               { Continued at EndComplete. }
            END;
      END;
END;
```

EndComplete is called when PPCEnd completes. It first checks for errors from PPCEnd. If the result is noErr, EndComplete either calls DoPPCInform to call PPCInform asynchronously again or puts the parameter block into gDoneQueue. If PPCEnd failed (the result is not noErr), EndComplete puts the parameter block into gDoneQueue. Any queued information is handled from the program's event loop.

```
PROCEDURE EndComplete (pbPtr: PPCParamBlockPtr);
BEGIN
   IF PPCEndPBPtr(pbPtr)^.ioResult = noErr THEN
      BEGIN { The PPCEnd call completed successfully. }
         IF NOT gQuitting THEN
            { Reuse the parameter block for another PPCInform. }
            DoPPCInform(PPCSessRecPtr(pbPtr))
            { Continued at DoPPCInform and then InformComplete. }
```

```
            ELSE
               { Drop the parameter block in the "done" queue for }
               { handling later. }
               Enqueue(QElemPtr(pbPtr), @gDoneQueue);
            { Dequeued by HandlePPCErrors. }
         END
      ELSE
         BEGIN { The PPCEnd call failed. }
            { Drop the parameter block in the "done" queue }
            { for handling later. }
            Enqueue(QElemPtr(pbPtr), @gDoneQueue);
            { Dequeued by HandlePPCErrors. }
         END;
END;
```

HandlePPCErrors is called regularly from the program's event loop. If gDoneQueue contains any parameter blocks, HandlePPCErrors removes the parameter blocks from the queue one at a time, checks to see what PPC call failed by inspecting the csCode field of the parameter block, and then handles the error condition appropriately. If the call that failed was PPCRead or PPCWrite, HandlePPCErrors calls PPCEnd asynchronously with EndComplete as the completion routine. Any errors returned by PPCEnd will be handled by EndComplete.

```
PROCEDURE HandlePPCErrors;
   CONST
   { PPC csCodes from async calls. }
      ppcOpenCmd = 1;
      ppcStartCmd = 2;
      ppcInformCmd = 3;
      ppcAcceptCmd = 4;
      ppcRejectCmd = 5;
      ppcWriteCmd = 6;
      ppcReadCmd = 7;
      ppcEndCmd = 8;
      ppcCloseCmd = 9;
      IPCListPortsCmd = 10;
   VAR
      pbPtr: PPCSessRecPtr;

BEGIN
   { Process any parameter blocks in the queue. }
   WHILE gDoneQueue.qHead <> NIL DO
      BEGIN
         { Get the PPCSessRec and remove it from the queue. }
         pbPtr := PPCSessRecPtr(gDoneQueue.qHead);
```

```
         IF Dequeue(QElemPtr(pbPtr), @gDoneQueue) = noErr THEN
            CASE PPCEndPBPtr(pbPtr)^.csCode OF
               ppcOpenCmd..ppcRejectCmd, ppcEndCmd..IPCListPortsCmd:
                  { For these calls, we'll just dispose of the }
                  { parameter block. }
                  DisposePtr(Ptr(pbPtr));
               ppcWriteCmd, ppcReadCmd:
                  BEGIN
                     { We need to call PPCEnd after read or write }
                     { failures to clean up after this session. }
                     PPCEndPBPtr(pbPtr)^.ioCompletion := @EndComplete;
                     { Error conditions are ignored in this procedure }
                     { because they are caught in EndComplete. }
                     pbPtr^.err := PPCEndAsync(PPCEndPBPtr(pbPtr));
                     { Continued at EndComplete. }
                  END;
            END;
      END;
END;
```

In this example of extending parameter blocks and using OS queues and call chaining, notice that asynchronous calls are chained together until an operation that can't be accomplished at interrupt time is necessary; then the extended parameter block is put into an OS queue where the main program can access it. Very few global variables are needed because OS queues are used to hold any data the main program code needs to access. Local variables aren't needed by the completion routines because the extended parameter block, PPCSessRec, holds everything the completion routines need.

DEBUGGING HINTS

Here are the top five debugging hints for writing asynchronous code.

Use Debugger or DebugStr calls and a low-level debugger. Because completion routines are called by the system, usually as a result of an interrupt, source-level debuggers don't work with completion routines. If you're having problems with a completion routine, first look at the parameter block used to make the asynchronous call. Look both before and after the call by using a Debugger or DebugStr call just before you make the asynchronous call and again at the beginning of the completion routine (remember, register A0 points to the parameter block).

Make sure parameter blocks are around for the life of the asynchronous call. The parameter block will have a whole new meaning if you forget and allocate a parameter block on the local stack, then make an asynchronous call with it, leave the current procedure or function, and reuse the stack for something new. There's nothing the system hates more than a completely bogus parameter block. If you

check your parameter block at completion time and the contents are different from what you expected, you've probably done this.

Don't reuse a parameter block that's in use. A parameter block passed to an asynchronous call is owned by the operating system until the asynchronous call completes. If you reuse the parameter block before the asynchronous call completes, at least one of the calls made with the parameter block will fail or crash the system. This can happen if you use a parameter block once from one place in your program and then forget and use it again from somewhere else.

Global parameter blocks should be avoided, because they're easy to reuse from several places within a program. If you keep your unused parameter blocks in an OS queue, you can safely claim one and reuse it anytime.

Avoid SyncWait. Does your Macintosh just sit there not responding to user events? Drop into the debugger and take a look at the code that's executing. Does it look like this?

```
MOVE.W   $0010(A0),D0
BGT.S    -$04,(PC)
```

That's SyncWait, the routine that synchronous calls sit in while waiting for a request to complete. Register A0 points to the parameter block used to make the call, offset $10 is the ioResult field of the parameter block, and SyncWait is waiting for ioResult to be less than or equal to 0 (noErr).

The ioResult field is changed by code executing as a result of an interrupt. If interrupts are disabled (because the synchronous call was made at interrupt time) or if the synchronous call was made to a service that's busy, you'll be in SyncWait forever. Take a look at the parameter block and where it is in memory, and you'll probably be able to figure out which synchronous call was made at interrupt time and which program made it.

Leave a trail of bread crumbs. There's nothing harder than reading chained asynchronous source code with no comments. You should always use comments to remind yourself where your chained call sequence goes. In the PPC code example given above, I left comments like "Continued at EndComplete" or "Dequeued by ProcessPPCData" to remind me where the chained call sequence will resume execution.

COMMON PROBLEMS AND THEIR SOLUTIONS

This section warns of some common problems and suggests ways to work around them.

TIME-CONSUMING TASKS AT INTERRUPT TIME

You may find a situation where a completion routine needs to perform some time-consuming task, but that task can be performed from interrupt-time code. This is a situation where the Deferred Task Manager may be useful. The Deferred Task Manager allows you to improve interrupt handling by deferring a lengthy task until all interrupts can be reenabled, but while still within the hardware interrupt cycle.

WAITNEXTEVENT SLEEP LATENCY

If you set your sleep time to a large value, maybe because you've been switched out, polling from the program's event loop may cause additional latency. The Process Manager's WakeUpProcess call, when available, can be used to shorten the time between when a completion routine queues a parameter block and when your program's event loop polls the queue header and processes the data in the parameter block. WakeUpProcess does this by making your program eligible for execution before the sleep time passed to WaitNextEvent expires.

The only parameter passed to WakeUpProcess is the process serial number of the process you want to wake up. You'll need to get your program's process serial number with the GetCurrentProcess function and add it to the extended parameter block used to call asynchronous functions:

```
{ Zero the process serial number. }
myPB.myPSN.highLongOfPSN := 0;
myPB.myPSN.lowLongOfPSN := 0;

{ Make sure the Process Manager is available. }
IF Gestalt(gestaltOSAttr, myFeature) = noErr THEN
   IF GetCurrentProcess(myPB.myPSN) = noErr THEN
      ; { Either we got the process serial number or it's still zero. }
```

The completion routine would use the process serial number (if available) to wake up your program immediately after queueing a parameter block:

```
{ Drop the parameter block in the "done" queue for handling later. }
Enqueue(QElemPtr(pbPtr), @gDoneQueue);

{ If we have a process serial number (myPSN <> 0), wake up the process. }
IF (pbPtr^.myPSN.highLongOfPSN<>0) OR (pbPtr^.myPSN.lowLongOfPSN<>0) THEN
   IF WakeUpProcess(pbPtr^.myPSN) = noErr THEN
      ; { Wake up the process. }
```

STACK SPACE AND CODE EXECUTING AT INTERRUPT LEVEL

Have you ever thought about the stack space used by interrupt code? Where does it come from? How much is available? Good questions.

The Deferred Task Manager is fully described in Chapter 6, "The Deferred Task Manager," in *Inside Macintosh: Processes*.•

When interrupt code (including completion routines) is called, it borrows space from whatever stack happens to be in use at the time. That means you have no control over the amount of stack space available, and so should use as little of the stack as possible.

At times, very little stack space is available, because some common Macintosh system calls temporarily use large portions of the stack. For example, some QuickDraw routines may leave as little as 1600 bytes of stack space and MoveHHi can leave as little as 1024 bytes of stack space under certain conditions. That's not a lot of space to borrow. If your interrupt code will call a routine that uses more than a few bytes of stack space, you should call the StackSpace function before calling that routine.

The typical symptom of using too much stack space is a random crash, because the memory you trash by overflowing the stack could belong to any program — including the Macintosh Operating System.

Here are the easiest ways to reduce the amount of stack space used by interrupt code:

- Don't use local variables. Either extend your parameter block to hold any variables needed by your completion routine or keep an OS queue of buffers that can be used by your completion routine.
- Try to keep the number of calls from your completion routine to other routines to a minimum. Each routine you call uses part of the stack to build a stack frame.

PROBLEMS WITH REUSING PARAMETER BLOCKS

There are three problems you may run into when you reuse a parameter block: unfortunate coercions, unfortunate overlap, and garbage in the parameter block.

Unfortunate coercions. Make sure parameter blocks are large enough for every use you'll put them to. For example, if you use a parameter block for both PPC Toolbox calls and File Manager calls, make sure the parameter block is large enough to hold any of the parameter blocks used by either manager. One way to do this is with a variant record:

```
variantPBRec = RECORD
      CASE INTEGER OF
      1: (PB: ParamBlockRec);       { File Manager parameter blocks }
      2: (cInfoPB: CInfoPBRec);
      3: (dtPB: DTPBRec);
      4: (hPB: HParamBlockRec);
      5: (cMovePB: CMovePBRec);
      6: (wdPB: WDPBRec);
      7: (fcbPB: FCBPBRec);
      8: (ppcPB: PPCParamBlockRec); { PPC Toolbox parameter block }
      END;
```

Unfortunate overlap. Don't assume variant record fields with the same name are in exactly the same place in the variant record. If they aren't, you'll run into problems with overlap. Check first and be sure.

Garbage in the parameter block. When reusing a parameter block, make sure data from the last use doesn't affect the next call. Always initialize all input fields. Many programmers go one step further by clearing the entire parameter block to zeros before initializing the input fields.

COMPLETION ORDER MIXUPS

Don't depend on a service being single-threaded (requests executed one at a time) or on requests being handled in the order they were made (first in, first out). The File Manager is single-threaded, but requests may not always be handled in the order they were made. The AppleTalk drivers allow multiple requests to execute concurrently.

If the order of completion is important, don't use concurrent calls — use chained calls. For example, if you write some data and then expect to read that data back, don't start an asynchronous write and then start an asynchronous read before the write completes. If the calls aren't handled in the order they were made, the read may complete before the write.

POLLING PROBLEMS

If your application that polls for completion works great when it's the current application, but slows down dramatically or stops when it's in the background, check for these common problems.

The canBackground bit. If you forget to set the canBackground bit in the SIZE resource, your application's event loop won't get called with null events while your application is in the background. If you're depending on null events for polling, your program won't poll while it's in the background.

Large WaitNextEvent sleep values. Did you crank up the sleep value passed to WaitNextEvent when your application received a suspend event? Talk about additional latency! This will do it if you're polling from the event loop.

What are other applications doing? Other applications can slow your event handling down by not calling WaitNextEvent regularly. If your polling from the event loop slows down because of that, there's not a lot you can do about it.

If your polling stops when another application is in the foreground, it could be that the other application isn't handling its update events. See Macintosh Technical Note "Pending Update Perils" for a description of this problem.

Polling from VBL tasks in an application's heap. VBL tasks in your application's heap are removed from the VBL queue during a context switch when your

application is switched out and are added to the VBL queue when your applicaton is switched back in. VBL tasks in the system heap are unaffected by context switches.

If you poll from a VBL task and don't want polling to stop when your application is switched out, make sure you put your VBL task in the system heap.

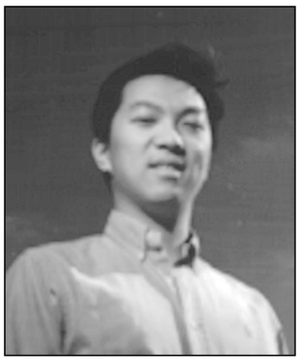
COMPLETION

There are many situations where synchronous calls work well. However, there are times when asynchronous calls must be used to prevent system deadlock or to let your program continue execution while waiting for time-consuming calls to complete. Understanding the material covered in this article should help you understand when to use asynchronous calls and give you the techniques needed to avoid the problems commonly encountered in code that executes asynchronously.

RECOMMENDED READING

- *Inside Macintosh* Volume II (Addison-Wesley, 1985), Chapter 6, "The Device Manager," Chapter 10, "The AppleTalk Manager," and Chapter 13, "The Operating System Utilities."
- *Inside Macintosh* Volume V (Addison-Wesley, 1986), Chapter 28, "The AppleTalk Manager."
- *Inside Macintosh* Volume VI (Addison-Wesley, 1991), Chapter 7, "The PPC Toolbox," Chapter 29, "The Process Manager," and Chapter 32, "The AppleTalk Manager."
- *Inside Macintosh: Files* (Addison-Wesley, 1992), Chapter 2, "The File Manager." Previously, *Inside Macintosh* Volume II, Chapter 4.
- *Inside Macintosh: Processes* (Addison-Wesley, 1992), Chapter 6, "The Deferred Task Manager." Previously, *Inside Macintosh* Volume V, Chapter 25 and Macintosh Technical Note #320, "Deferred Task Traps, Truths, and Tips."
- Macintosh Technical Notes "MultiFinder Miscellanea" (formerly #180), "Setting and Restoring A5" (formerly #208), "NuBus Interrupt Latency (I Was a Teenage DMA Junkie)" (formerly #221), and "Pending Update Perils" (formerly #304).

THANKS TO OUR TECHNICAL REVIEWERS
Scott Boyd, Neil Day, Martin Minow, Gordon Sheridan•



SOMEWHERE IN QUICKTIME

TOP 10 QUICKTIME TIPS

JOHN WANG

To inaugurate this new column on QuickTime, we'll take a look at ten useful tips for QuickTime application developers. This is certainly not an exhaustive list, but it *is* an important one.

Here's the list:

10. Working around data reference limitations.
9. Using GetMovieNextInterestingTime.
8. Not calling ExitMovies.
7. Getting a movie's unscaled size.
6. Avoiding the Movie Toolbox when using the standard movie controller.
5. Prerolling a movie for improved playback.
4. Using CustomGetFilePreview with custom dialogs.
3. Conditionally registering a component that requires a hardware device.
2. Detaching a movie controller properly.
1. Calling MaxApplZone from every application.

Some of the tips describe pitfalls that need to be avoided, while others are simply clarifications. Let's take a closer look at each one.

10. Working around data reference limitations.

A current limitation of QuickTime is that each media can have only one data reference to a media data file. This isn't a problem except when you start cutting and pasting between tracks that refer to different media. You'll then be required to copy the media data from one media data file to another. For example, InsertTrackSegment will copy media data between media if the tracks refer to different media.

Calls like GetMediaDataRefCount, AddMediaDataRef, and GetMediaDataRef will reflect the "one data reference" limitation by only accepting index values of 1. You can't replace an existing media data reference in QuickTime 1.0, but you can in QuickTime 1.5, with a new call, SetMediaDataRef. Using this routine is a common way of manually resolving media data references that may have been moved by an application. For example, if you move a movie data file onto a different volume, you can update the alias using the Alias Manager and update the data reference for the movie with SetMediaDataRef.

9. Using GetMovieNextInterestingTime.

Since QuickTime is time based rather than frame-number based in the way it deals with temporal video data, a common question is how to get information about movie frames, such as frame rate. The answer is to use GetMovieNextInterestingTime. This function allows you to step quickly and easily through interesting times in a movie. For example, for an estimate of the frame rate of a movie, you could use GetMovieNextInterestingTime to count the total number of frames in the movie and divide it by the total duration of the movie. Likewise, you could use GetMovieNextInterestingTime to identify the 600th frame in a movie. Since the internal data structure of QuickTime movies is optimized for accessing this type of information, GetMovieNextInterestingTime and the other GetNextInterestingTime calls are very efficient.

8. Not calling ExitMovies.

One recommendation that contradicts QuickTime 1.0 documentation is that applications should not call

JOHN WANG (AppleLink WANG.JY) is enjoying his youth in the playpen of the Printing, Imaging, and Graphics (PIGs) group in Developer Technical Support at Apple. When he's not engaged in piglet activities, he can be found on a golf course or hogging the road with his Mazda Miata. No one has trouble identifying John's car, since he often cruises the California highways with his dog, Skate. In return, Skate promises to drive safely.•

ExitMovies before quitting. QuickTime calls this function at ExitToShell time, and it's safer to allow QuickTime to release private storage and component connections at that time. This prevents problems such as closing components in the wrong order. For example, the proper way to clean up after a movie that uses a standard movie controller is to dispose of the movie controller first, and then dispose of the movie. If done in the reverse order, there may be adverse consequences.

7. Getting a movie's unscaled size.

An application should save the movie box obtained with GetMovieBox when a movie is loaded so that it can retrieve the intended offset and scaling of the movie for playback. However, some applications may also want to get the unscaled size, and there isn't an intuitive way to get it. Since the programmatical effect of calling SetMovieBox is that the movie matrix is changed to reflect the new offset and scaling, you can easily get the movie box for an unscaled movie by setting the movie matrix to the identity matrix; using the utility routine SetIdentityMatrix along with SetMovieMatrix accomplishes this. Then GetMovieBox will return the unscaled size and offsets.

However, there's a loophole. If a track inside the movie is scaled, there may still be scaling in playback since QuickTime supports transformation matrices for movies and for tracks within a movie. Therefore, when working with scaling, applications need to pay attention not only to the movie's scaling but to the tracks' scaling as well.

6. Avoiding the Movie Toolbox when using the standard movie controller.

When using the standard movie controller, you should almost never use any Movie Toolbox routines that control movie playback or change movie characteristics. For example, it would be a mistake to call StartMovie to start playing a movie. Instead, use the movie controller equivalent, MCDoAction with mcActionPlay. Calling StartMovie directly causes the movie to play but with the controller's button in the pause state, not reflecting that the movie is playing back. Similarly, to set looping, you would use MCDoAction with mcActionSetLooping. Bypassing the movie controller by using the Movie Toolbox routines directly on a movie controlled with a movie controller can have dire consequences. As an example, if you set looping for a movie before creating a controller with NewMovieController, you'll cause the Macintosh to crash. Don't let it happen to you!

5. Prerolling a movie for improved playback.

Prerolling can improve playback performance by allowing QuickTime to do preliminary initialization. Since PrerollMovie is passed the movie time and rate, it can fill buffers and caches optimally to prevent initial stuttering. Normally, QuickTime automatically prerolls the movie for you. For example, if you call StartMovie, you don't need to also call PrerollMovie, since StartMovie prerolls the movie for you. The QuickTime 1.5 documentation describing the StartMovie call states this clearly. Likewise, the standard movie controller is optimized to preroll whenever the user starts a movie with the keyboard or mouse. If you call PrerollMovie in these situations, the second PrerollMovie is redundant and will simply waste time. In all other cases, prerolling by calling PrerollMovie is recommended before initiating playback. For example, you should call PrerollMovie before SetMovieRate.

4. Using CustomGetFilePreview with custom dialogs.

If you use CustomGetFilePreview with custom DLOG and DITL resources, you should be aware of a bug with the System 7 pop-up menu CDEF: pop-up menus used in conjunction with black-and-white grafPorts are shifted to the wrong location within the dialog box. The simple workaround is to force the dialog to be a cGrafPort by adding a 'dctb' resource with the same ID as the DLOG and DITL resources. You can easily create a 'dctb' resource with ResEdit by selecting the Custom color button in the DLOG resource template window. For more information on the 'dctb' resource, see the Dialog Manager chapter in *Inside Macintosh: Macintosh Toolbox Essentials* (or in *Inside Macintosh* Volume V).

3. Conditionally registering a component that requires a hardware device.

If you write a component that requires a hardware device, you should set the wantsRegisterMessage flag to give your component an opportunity to verify that the specific hardware is properly installed. If the hardware isn't available, you can then indicate to the Component Manager that you don't want the component registered. The register routine, called with selector kComponentRegister, should return FALSE if it does want to be registered and TRUE if it doesn't.

One thing to be aware of is that even during registration, the component will be opened with OpenComponent and closed with CloseComponent. Therefore, you can expect OpenComponent before the ComponentRegister routine is called, and CloseComponent after ComponentRegister is called.

For example, if you have a 'vdig' that works with a NuBus video digitizer card, each time that OpenComponent is called you can check whether the hardware is correctly installed, and then return that status when ComponentRegister is called by the Component Manager.

2. Detaching a movie controller properly.

If you want to place the standard movie controller in a different window or location from its usual placement directly below the movie, you must detach the movie controller. Follow these steps:

1. First bring up the controller by calling NewMovieController with the flag mcNotVisible so that the controller is initially invisible. If you don't do this, the application will momentarily display the controller in the wrong location.
2. Call MCSetControllerAttached with FALSE to detach the controller.
3. Call MCSetControllerPort to move the controller to a different port if you want to place it in a different window. If you only want to move the controller in the same window as the movie, you don't have to call MCSetControllerPort.
4. Call either MCPositionController or MCSetControllerBoundsRect to move the controller to the new location in the port.
5. Call MCSetVisible to display the controller.

The movie will remain in whatever port it was assigned to using SetMovieGWorld. If MCSetControllerPort isn't called (step 3), the controller will remain assigned to the movie's port when NewMovieController is called.

For example:

```
SetMovieGWorld(myMovie, (CGrafPtr) myWindow, 0);
mcMC = NewMovieController(myMovie, &movieBounds,
            mcTopLeftMovie + mcNotVisible);
MCSetControllerAttached(mcMC, FALSE);
MCSetControllerPort(mcMC, myOtherWindow);
MCPositionController(myMC, &movieBounds,
            &newControllerRect, mcTopLeftMovie);
MCSetVisible(myMC, TRUE);
```

1. Calling MaxApplZone from every application.

Not calling MaxApplZone in an application is the reason why many simple QuickTime playback applications play back movies poorly. Because the Memory Manager grows the heap only if there isn't any purgeable or free space left, QuickTime doesn't have the space it needs to play back a movie optimally. Since there's no penalty or drawback for calling MaxApplZone, all applications should call the routine during initialization. In fact, MaxApplZone should be your first Macintosh Toolbox call, because initializing QuickDraw and other managers could allocate memory.

We hope these tips will help you avoid some of the most common pitfalls of QuickTime development. With so many developers writing QuickTime applications and adding QuickTime support into existing applications, we want the journey to be as smooth as possible. We'll keep you updated and informed by continuing to bring you insightful tips and details about QuickTime in this column. Watch for it!

For more information on the Component Manager, see the QuickTime or System 7.1 documentation on this subject, and see Gary Woodcock and Casey King's article, "Techniques for Writing and Debugging Components," in *develop* Issue 12.•

Thanks to the developers who have made this list possible and to Bill Guschwan, Peter Hoddie, and Guillermo Ortiz for reviewing this column.•

INSIDE QUICKTIME AND COMPONENT-BASED MANAGERS

Intercepting the processing of a QuickTime routine enables you to debug the routine, use the routine in new ways, and better understand QuickTime architecture. To intercept the routine, you need to know something about its low-level implementation. This article discusses the low-level implementation of QuickTime routines, and also describes tools and programming techniques that can be used to debug, modify, and analyze QuickTime routines. Some of these techniques take advantage of the Component Manager, and their usefulness will extend beyond QuickTime as future managers capitalize on components.

BILL GUSCHWAN

As QuickTime routines pass through some common locations, they're accessible to your application or to a debugger. A QuickTime routine begins with its function name, as used in your application and defined in the interface files. It usually compiles as an A-trap and maybe some assembly glue. The routine may call other Macintosh routines, be affected by global data structures, pass through a grafPort's bottleneck, or pass through a component's main function. Because you have access to these locations, you can intercept the processing of the routine, perform your own special processing, and then allow the normal execution of the routine to continue.

This article's examples use MacsBug and TMON Pro (TMON Professional v. 3.0.1 from Icom Simulations, Inc.) to intercept and analyze routines. The tools discussed create resources for both debuggers, though in some situations you'll want to use one debugger over the other. For example, the language extensibility of TMON Pro's built-in assembler provides capabilities that other debuggers don't provide. Now let's get into the practical aspects of analyzing and debugging QuickTime routines.

QUICKTIME A-TRAPS

An A-trap is a two-byte opcode that always begins with the hexadecimal numeral A. The remaining 12 bits in the opcode identify the particular routine you're calling,

BILL ("ANGUS") GUSCHWAN describes Angus as an identity cocktail in the sky. If his favorite philosophers, character, and author were alive today, we can imagine what they might say about the young man and the sky. Gottlieb Frege: "A setting sun indicates the object, sun. But the sun also rises. Just as a night in the forest, mountains in springtime, and a walk in the rain convey solitude, each sense adds knowledge to the meaning of the sun. Thus, Angus does not singularly denote Bill Guschwan, but rather indicates a sense of him." Ludwig Wittgenstein: "Bullfighting is an analogy for life. Angus represents the bull, whereas language represents the toreador's red cape. Thus, Angus perishes if he trusts it, and destroys if he ignores it." Andromache: "As a young Indian identifies with soaring hawks, young Angus identifies with the

along with other information about the call. A-traps interrupt the normal processing of the CPU and cause it to jump through a low-memory vector to the trap dispatcher. The trap dispatcher examines the bit pattern of the opcode to determine the actual location of the Macintosh routine in memory, and then jumps to it. Almost all Macintosh Toolbox routines use the A-trap mechanism to jump to their code.

In the early days of the Macintosh, there was one routine name per A-trap, but the number of routines increased so dramatically that a second mechanism was introduced to avoid exhausting all the A-traps. This mechanism uses the normal A-trap mechanism to identify a grouping of routines (usually defined by a specific manager) and uses selectors located on the stack or in a register to identify the specific routines within the grouping. QuickTime uses only four A-traps:

- 0xAAAA: Movie Toolbox
- 0xA82A: Component Manager
- 0xAAA3: Image Compression Manager
- 0xABC2: Matrix routines

Using four A-traps for over 500 routines is possible because the interface glue can push routine selectors into registers or onto the stack. QuickTime picks the routine it needs to execute from the value of the selector. For example, with the Movie Toolbox, QuickTime uses a word in the D0 register. So 0x303C and xxxx (the two-byte selector) appear before the A-trap in the Movies.h file. This disassembles into MOVE.W #$xxxx, D0. If you want to find out what other opcodes mean, try using the TMON Pro assembler as described in "TMON Pro Assembler Demo."

On a separate note, components implement routines through selectors as well. In some ways, a component is not unlike an A-trap. The ramifications of this are discussed later in the section "Bottlenecks."

TRAPPING COMPILED APPLICATIONS

A QuickTime routine's A-trap provides a common location that your debugger can interact with. Traditionally, Macintosh developers have used MacsBug to investigate the flow of A-traps in compiled applications. Knowing the sequence of A-traps needed to implement specific functionality provides invaluable information exceeding the scope of even the best documentation.

Let's see what happens when we take the simple QuickTime debugging approach of breaking on the four A-traps. For example, start with the 0xAAAA trap. If you perform an "atb _AAAA" and run MoviePlayer, MacsBug is continually invoked. You can use the debugger to see the selector value that identifies the routine, but unless you have the interface files in front of you or you memorize the selector values, you won't be able to tell which QuickTime routine is being called. You can probably

lost generation of somnambulating dogcows. As an Indian peasant links with god via the farm tools in the hands of a Buddhist statue, Angus links with god via the TMON Pro manual in the hands of a Zimmerman statue." Ernest Hemingway: "OK. Sure, Angus. Anyone for a martini cocktail? With a twist."•

TMON PRO ASSEMBLER DEMO

TMON Pro has an assembler/disassembler built in. You can enter TMON Pro, type hexadecimal machine code, and watch as it's disassembled into assembly. To do this, you need to make use of TMON Pro's typed windows, which provide alternative views of the same location in memory. So, if you anchor an Assembly window and a Memory window at some safe location in memory, you can type machine code in the Memory window and watch the numbers translate into the assembly routines in the Assembly window.

TMON Pro sets aside an area of memory for you to play with, identified by the variable PlayMem. Here's a useful alias that you can install in your TMON script (it assumes you use the script provided with TMON Pro):

```
alias PlayTime,
"TopWind .10 ∂n New Memory HereHP, :Δplaymem ∂
BottomWind .6 ∂n New Assembly HereHP,Δplaymem ∂
Open Registers #1=#0"
```

Now you can type "PlayTime" at the command line and have a safe area in memory for exploring the TMON Pro assembler. The PlayTime alias anchors the two windows to the same place in memory and swaps out the registers so that you don't harm them while you play (see Figure 1).

```
 [] Disassembly from $00076670          []
00076670: 'PlayMem':_EnterMovies
00076676: 'PlayMem'+$0006: MOVE.W       #$0002,D0
0007667A: 'PlayMem'+$000A: _AAAB
0007667C: 'PlayMem'+$000C: ORI.B        #$00,D0
00076680: 'PlayMem'+$0010: ORI.B        #$00,D0
00076684: 'PlayMem'+$0014: ORI.B        #$00,D0
00076688: 'PlayMem'+$0018: ORI.B        #$00,D0
0007668C: 'PlayMem'+$001C: ORI.B        #$00,D0
00076690: 'PlayMem'+$0020: ORI.B        #$00,D0
```

```
 [] Memory from :$00076670              []
00076670: 'PlayMem'      : 303C0001  AAAA303C  0002AAAB  00000000 0<..™™0<..™´....
00076680: 'PlayMem'+$0010: 00000000  00000000  00000000  00000000 ................
00076690: 'PlayMem'+$0020: 00000000  00000000  00000000  00000000 ................
000766A0: 'PlayMem'+$0030: 00000000  00000000  00000000  00000000 ................
000766B0: 'PlayMem'+$0040: 00000000  00000000  00000000  00000000 ................
000766C0: 'PlayMem'+$0050: 00000000  00000000  00000000  00000000 ................
000766D0: 'PlayMem'+$0060: 00000000  00000000  00000000  00000000 ................
000766E0: 'PlayMem'+$0070: 00000000  00000000  00000000  00000000 ................
000766F0: 'PlayMem'+$0080: 00000000  00000000  00000000  00000000 ................
```

Alternative views of the same location in memory

Figure 1
TMON Pro Windows

memorize a few routines like EnterMovies, which has a selector value of 1. You could even record all the A-trap routines (using the **atr** command), print to a file, and compare the traps against the interface files. However, these methods leave a lot to be desired.

Because there's no one-to-one correspondence between A-traps and routines, you need some tools to facilitate trapping QuickTime applications. To take advantage of trapping compiled applications, you'd like to be able to do the following:

- Set the A-trap break on the routine name.
- Easily identify the routines in the debugger.

USING 'MXBM' RESOURCES

You can set A-trap breaks on QuickTime routine names by creating MacsBug macros in the form of 'mxbm' resources. Unfortunately, MacsBug doesn't ship with the 'mxbm' resources for QuickTime, and creating those resources by hand would be tedious at best. So I wrote debugit, an MPW tool that converts standard Macintosh C headers into the resources. The tool and the 'mxbm' resources that are needed to set QuickTime A-trap breaks are on the *Developer CD Series* disc and the *QuickTime Version 1.5 for Developers* disc. (Also supplied are the 'mxbm' resources for several other managers that use A-traps with routine selectors.) You simply place the resources in your Debugger Prefs file using a resource editor and reboot.

Using MacsBug in this way is still limited because even though you can break on a routine name, the names of the QuickTime routines aren't displayed when you're in MacsBug — only the assembly code is displayed.

USING A TMON PRO USER AREA

You saw (in "TMON Pro Assembler Demo") how you can type machine code in TMON Pro and watch it disassemble. While this is fun, its practical use for developers is limited. The real power of the TMON Pro assembler comes from the extensibility of its language. With a little work, you can use TMON Pro to both break on routine names and display routine names instead of assembly code in the debugger.

To extend the vocabulary of TMON Pro's interactive assembler, you need to create TMON Pro assembler macros for the A-traps and glue, which TMON Pro disassembles into the QuickTime function name. TMON Pro looks many instructions ahead to disassemble the A-trap and glue into the routine name. If you create the requisite 'Asm ' resources, the TMON Pro Assembly window can display code like

```
MOVE.W #1,D0
_AAAA
```

as follows:

```
_EnterMovies
```

If you create the proper aliases ('mxbm' resource equivalents), you can set A-trap breaks on QuickTime routine names as well.

Creating the 'Asm ' resources manually is impractical, so I modified debugit to create both the assembler macros and the aliases for setting breaks on the QuickTime routine names from a Macintosh C interface file. To load the 'Asm ' resources into TMON Pro, you also need to create a TMON Pro user area to hold the 'Asm ' resources (see "Creating Debugging Tools"). To keep the resources and aliases in one location, you place the aliases in the data fork of the TMON Pro user area. TMON Pro looks there when it's loading scripts. To use the QuickTime Angus User Area (which is on the *Developer Series CD* disc), just drop it in your TMON folder and reboot. Remember, this user area is large and contains an alias for every QuickTime routine. But it's easy to pull it out if you want to run stealthily.

CREATING DEBUGGING TOOLS

Although the QuickTime Angus User Area and 'mxbm' resources are included on the *Developer Series CD* disc, instructions for creating them are given here to show how simple it is. You could create resources for other managers using the same technique. The CD includes a script that uses the following commands to create the MacsBug and TMON Pro resources for QuickTime.

MAKING AN ANGUS USER AREA

To create a debugging user area for TMON Pro you need to have TMON Pro installed, because the script will automatically place the user area in your TMON Folder. In addition, you need to do the following:

- Place the MakeUserArea script in your MPW Scripts folder.
- Place the debugit MPW tool (on the CD) in your MPW Tools folder.
- Place the TMONTypes.r and Macsbug.r files in your MPW RIncludes folder.
- Place the User Area Template (on the CD) in your current directory.

With the tools properly stored, you can create the QuickTime Angus User Area with the following command:

```
makeuserarea {CIncludes}"Movies.h" ∂
   {CIncludes}"ImageCompression.h" ∂
   {CIncludes}"Components.h" ∂
   {CIncludes}"QuickTimeComponents.h" ∂
   {CIncludes}"MediaHandlers.h"
```

MakeUserArea is a script that uses the Rez, C, and debugit tools, so you can alter its behavior fairly easily. Be sure to use the script with the managers of your choice!

MAKING 'MXBM' RESOURCES

To make 'mxbm' resources, you need to place the debugit tool in your MPW Tools folder, Macsbug.r in your MPW RIncludes folder, the MakeMxbm script in your MPW Scripts folder, and a Debugger Prefs file in your System Folder. Here's how to make the 'mxbm' resources for Movies.h:

```
makemxbm {CIncludes}Movies.h MoovDispatch 128
```

With the QuickTime Angus User Area you can set breaks as you do with 'mxbm' resources in MacsBug. Just type the routine name without the underscore at the command line (type Command-space to invoke the command line). By default, typing the name of the QuickTime routine sets an intercept action, or break, for the A-trap. You can also specify the other four trap actions by using the trap action keywords after the QuickTime routine name. For example, to turn on a heap trap action every time EnterMovies is called, type

```
entermovies heap
```

You can also turn off trap actions from the command line. So, for example, if you type "findnextcomponent," you can cancel it with "findnextcomponent nointercept." You can shorten your commands by creating a macro such as

```
macro ni,"nointercept"
```

Several useful macros are included as a separate script on the *Developer Series CD* disc. See the TMON Pro reference manual for more information on using macros.

When you break into the debugger and look in the Memory window, TMON Pro's interactive assembler uses the 'Asm ' resources from the resource fork of the user area to interpret the assembly code and display routine names. Now you have the tools you need to easily watch the flow of QuickTime routines in a compiled application (see Figure 2).

SETTING A-TRAP BREAKS ON COMMON ROUTINES

As mentioned earlier, a Macintosh Toolbox routine's code is located via the A-trap vector, which provides a convenient location for interaction with a debugger. While watching the flow of A-traps can help you understand a manager, sometimes microscopic detail is needed to understand a specific routine. Historically, Macintosh developers have used MacsBug to investigate internal routines of Macintosh A-traps and provide keen insight where *Inside Macintosh* leaves off. This is usually done by setting A-trap breaks on routines called by the routine being investigated.

BREAKING ON COMMON RESOURCE MANAGER ROUTINES

It may seem too obvious to mention that Macintosh routines use other Macintosh routines, but it's a crucial debugging concept. Given a routine and its functionality, good Macintosh programmers can make excellent guesses as to which other routines it uses. For example, FlattenMovie calls an internal version of FlattenMovieData.

Because a movie is the significant data structure introduced with QuickTime, let's look at new movie calls (NewMovie, NewMovieFromFile, NewMovieFromHandle, NewMovieFromDataFork, and NewMovieFromScrap). Setting A-trap breaks on Macintosh routines is best done with a small speedy debugger — like MacsBug. So

```
   [] newmoviefromfile
```

```
 [] Memory from : PC                                                    []
00D4F64C:P     'OpenTheMovie'+$00BA: _MovieDispatch
00D4F64E:      'OpenTheMovie'+$00BC: MOVE.W  (A7)+,D0
00D4F650:      'OpenTheMovie'+$00BE: EXT.L   D0
00D4F652:      'OpenTheMovie'+$00C0: MOVE.L  D0,D7
00D4F654:      'OpenTheMovie'+$00C2: MOVEQ   #`80,D0                        ;'Ä'
00D4F656:      'OpenTheMovie'+$00C4: CMP.L   D7,D0
00D4F658:      'OpenTheMovie'+$00C6: BNE.S   ^$00D4F668          ;'OpenTheMovie'+$D6
00D4F65A:      'OpenTheMovie'+$00C8: TST.B   D5
00D4F65C:      'OpenTheMovie'+$00CA: BNE.S^  $00D4F668           ;'OpenTheMovie'+$D6
00D4F65E:      'OpenTheMovie'+$00CC: ORI.L   #$00000002,D6
```

```
[] Disassembly from  PC-4                                               []
00D4F648:      'OpenTheMovie'+$00B6: _NewMovieFromFile
00D4F64E:      'OpenTheMovie'+$00BC: MOVE.W  (A7)+,D0
00D4F650:      'OpenTheMovie'+$00BE: EXT.L   D0
00D4F652:      'OpenTheMovie'+$00C0: MOVE.L  D0,D7
00D4F654:      'OpenTheMovie'+$00C2: MOVEQ   #`80,D0                        ;'Ä'
00D4F656:      'OpenTheMovie'+$00C4: CMP.L   D7,D0
00D4F658:      'OpenTheMovie'+$00C6: BNE.S   ^$00D4F668          ;'OpenTheMovie'+$D6
00D4F65A:      'OpenTheMovie'+$00C8: TST.B   D5
00D4F65C:      'OpenTheMovie'+$00CA: BNE.S   ^$00D4F668               ;'OpenTheMovie'+$D6
00D4F65E:      'OpenTheMovie'+$00CC: ORI.L   #$00000002,D6
```

```
[] Traps                                                                []
«rhSi» _WaitNextEvent
«rhSi» _GetNextEvent.._EventAvail
«rhsi» ..:  $2AA^<PC&&PC<$130^
«rhsI» _MovieDispatch:  D0.W==$F0
```

```
 [] Memory from :$00000910                                              []
00000910:         'CurApName'      : "MoviePlayer"
```

Intercept trap for NewMovieFromFile

QuickTime routine name

Figure 2
The Flow of QuickTime Routines in TMON Pro

let's use MacsBug to find out how QuickTime loads its data. As you probably know, the data structure for a movie is undocumented. While any type of manipulation with the movie can be done with the Movie Toolbox, leaving the movie data structure undocumented can cause some confusion as to how a movie actually works. In fact, the movie on the disk is different in structure from the movie in memory. While the movie on disk is documented, the movie in memory is not, which lets the QuickTime team change the loaded movie without affecting your application. Keep that in mind as you begin investigating the exact nature of the movie in memory.

The target application for this investigation is MoviePlayer because it calls the various new movie routines. MoviePlayer was created by the QuickTime team, and it's widely distributed. If you launch the application and choose Open from the File menu, you're presented with the CustomGetFilePreview dialog box.

To look at the internals of an individual routine, you need to drop into the debugger before executing the routine. Simply set your traditional A-trap break and go:

```
atb newmoviefromfile; g
```

Next, open a movie that uses a 'moov' resource. Now you're ready to investigate NewMovieFromFile's use of internal routines. Since QuickTime uses the Resource Manager, you'll set a break on GetResource and expect NewMovieFromFile to load the 'moov' resource from a file. In MacsBug, set a break on the condition:

```
atb getresource (sp+2)^='moov'; br pc+2
```

This command lets you check for all the calls that NewMovieFromFile makes to GetResource that load a 'moov' resource. Watch for one of the following messages in the debugger:

```
Breakpoint at address routinename
A-Trap break at address routinename
```

If you see the first message before the second, you know that NewMovieFromFile doesn't use GetResource. As you'll see, GetResource is not called.

But you don't need to give up on the GetResource idea. Some A-traps have variations, which makes it difficult to guess which routine is called. Two obvious variations of GetResource are Get1Resource and Get1IndResource. NewMovieFromFile can be passed nil for the resource ID, which means it probably loads the first 'moov' resource. With this theory in mind, break into NewMovieFromFile again, and this time set the break on Get1xResource instead of GetResource (Get1xResource is the MacsBug equivalent of Get1IndResource):

```
atb get1xresource (sp+2)^='moov'; br pc+2
```


To easily read the type of resource in the upper left corner of MacsBug, try executing the command "show 'sp+2' a". The **a** parameter lets you view the information in ASCII, and the single quotation marks tell MacsBug to anchor the status region to the changing location of the stack pointer. In TMON Pro, use the command "Δ(sp+2)" in a Memory window.•

When you leave MacsBug, you'll get an A-trap break and thus know how NewMovieFromFile loads the movie.

Unfortunately, breaking on GetResource works for only one of the five new movie calls. You don't get a break with NewMovie, because the call is similar to a NewWindow call and doesn't bring in a resource. You may get a break with a NewMovieFromFile call, since it does bring in the 'moov' resource from the file. It's similar to a GetNewWindow call, but it may break on Get1IndResource or Get1Resource, depending on whether you supplied a resource ID to the call. NewMovieFromHandle and NewMovieFromDataFork will not break, because a movie doesn't have to be stored in a resource. You don't get a break for NewMovieFromScrap, because it loads the movie directly from the scrap.

As you've seen, although breaking on GetResource can provide some insight, it's limited in what it can tell you about the general class of new movie calls. Breaking on GetResource showed you how the new movie calls differ in their methods of loading the data. However, it didn't show how they implement their common behavior. Their similar names indicate that the calls exhibit similar behavior in loading a movie into memory. While it's true you can break on the loading of code resources, and even code resources of different types (WDEF, CDEF, INIT), you have limited information to differentiate one code resource from another (other than by the resource type). Thus, we turn to techniques for breaking on component routines.

BREAKING ON COMMON COMPONENT MANAGER ROUTINES

Components consist of a set of routines that implement a specific type of functionality. To identify the exact nature of the functionality, a component has an associated 'thng' resource. (At one point in their evolution, components were called "things.") The 'thng' resource stores a reference to the component code, a ComponentDescription record, string resources, and an icon resource. The ComponentDescription record identifies the type of functionality that the component's set of routines implements; for example, a media handler component is identified by the OSType 'mhlr' in the type field of the ComponentDescription record. Thus, components make it possible to break on the loading of functionality.

Components are identical to code resources, except that a component uses an extended resource specification in the form of the 'thng' resource. Normal resources use a resource type and ID for their resource specification. Because a component consists of a typed code resource and a 'thng' resource, you can use the traditional GetResource techniques on components, but in newer and better ways.

So let's exploit QuickTime's use of components. QuickTime depends on over 50 components. The best call to break on is FindNextComponent, which queries the Component Manager for components and returns a reference to a component. It's consistently called by applications that need a component, and its parameters contain extra information about the component. Breaking on OpenComponent isn't as useful

Breaking on internal A-traps assumes that QuickTime uses the A-trap mechanism. A later example illustrates how this assumption can affect your investigations.•

For more information on components, see the QuickTime or System 7.1 documentation on the Component Manager, and see Gary Woodcock and Casey King's article, "Techniques for Writing and Debugging Components," in *develop* Issue 12.•

because you have no simple way of identifying the component type. You break on FindNextComponent just as you do with GetResource:

```
atb findnextcomponent
```

The first field of a ComponentDescription record is the component type. Since it's the last parameter pushed on the stack, you can anchor a dereferenced stack pointer to the upper left corner of MacsBug:

```
show 'sp^' a
```

By watching the status region, you can see which components QuickTime loads and when they're loaded. This helps you understand the internal behavior of a routine. Alternatively, in TMON Pro, you could anchor a Memory window to a dereferenced stack pointer, as shown in Figure 3.

Unfortunately, QuickTime doesn't always call the A-trap mechanism for some internal routines. A notable example is OpenDefaultComponent, which may not call FindNextComponent via the A-trap mechanism. It can use a direct dispatch mechanism, which helps speed up QuickTime. One solution to this problem is to set an A-trap break on OpenDefaultComponent as well as FindNextComponent. Another solution is to use the **thing** dcmd and an A-trap break on OpenComponent. Even though with OpenComponent you have no simple method of identifying the type of component, at least OpenComponent must always be called for any component that's opened. The **thing** dcmd lets you find out what type of component is loaded. It lists all components registered with the Component Manager and, in the far left column, lists the number of instances.

Let's consider the NewMovieFromFile example again. You break on NewMovieFromFile, and then execute the **thing** dcmd to see what components are loaded, remembering particularly the number of instances. Next, you break on OpenComponent, step over it, and invoke **thing** again. You can easily notice the change in instances for the 'clok' component. This technique may be a little more cumbersome, but because QuickTime sometimes bypasses the trap dispatch mechanism, it's more accurate.

As more Macintosh Toolbox managers rely on components, you'll find trapping on typed functionality to be invaluable to your understanding of that manager. Debugging techniques that you've used with the Resource Manager can be used successfully with the Component Manager.

DYNAMIC STATE INFORMATION

You've seen how debuggers can interact with A-traps to provide valuable information about QuickTime routines. Now let's leave the realm of debuggers and focus on the

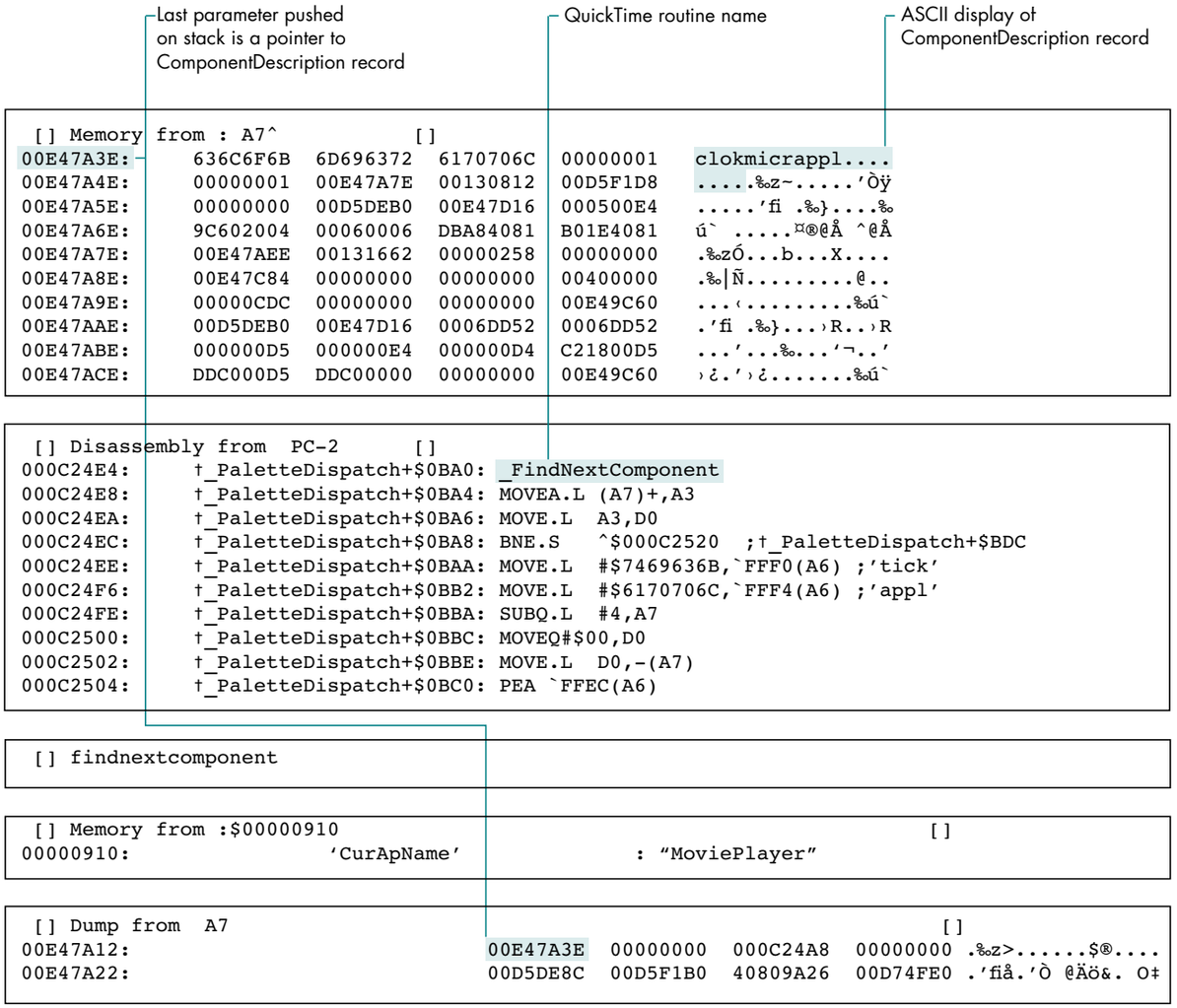


Figure 3
Breaking on Component Routines With TMON Pro Debugging Tools

interaction of global data structures and QuickTime routines. The Macintosh uses state information extensively to build simulations of real-world environments. QuickDraw's grafPort provides a familiar example — it contains state information to provide a consistent context for graphics operations. But it can trip you up if you're not aware of that context.

With that in mind, let's continue our investigation of QuickTime routines. Go back to MoviePlayer and set the breaks again on NewMovieFromFile. Then use the technique described in the previous section to find out which components are loaded. NewMovieFromFile first loads a 'clok' component. This is probably part of a NewTimeBase call. Testing this guess by breaking on NewTimeBase shows that the TimeBase is created dynamically — it's not a static part of a movie file format. What does it mean that all NewMovieFromFile calls load a TimeBase?

QuickTime adds its own context in the form of dynamic state information. By default, a movie generates a TimeBase. Just as GrafPort supplies a data structure for graphical state information, TimeBase provides a data structure for time information. Any time can be autonomously specified by a time base, time scale, and time value, which are grouped in a convenient data structure called TimeRecord.

If you work with QuickTime a lot, you'll notice that you seldom use TimeRecord. It seems odd until you realize that if you use a movie, you already have a default TimeBase supplied. There's no point to filling out a TimeRecord structure. There are easy calls to get the movie time scale (such as GetMovieTimeScale), and you usually specify a time value. Developers often forget the time context and make redundant calls. For example, developers forget that StartMovie calls SetMovieRate with the movie's preferred rate, and call both StartMovie and SetMovieRate. For movies, don't forget the time context. (This is not to say that TimeRecord is useless; when you don't have a movie and need to specify a specific time, TimeRecord comes in handy.)

If you continue breaking on component routines, you'll see that after loading a 'clok' component, NewMovieFromFile dynamically loads its media handlers. The Movie Toolbox doesn't know how to interpret media: it leaves that task to the media handlers. (Media handlers are discussed later under "Component Bottlenecks.") A movie is a dynamically loaded series of components. As a further exercise for breaking on component routines, try looking at the components that CustomGetFilePreview uses.

BOTTLENECKS

Some programming techniques allow you to alter Macintosh routines. QuickTime relies extensively on QuickDraw, and QuickDraw uses bottlenecks to implement its routines' functionality. Bottlenecks are commonly used in two ways:

- You can observe the behavior of an entire group of routines by replacing one bottleneck routine with your own. Most commonly, you would put a Debugger statement in it.
- You can gain access to information at a lower level and before it's been worked on. You can either change this information or use it for other purposes.

Time bases are discussed in "Time Bases: The Heartbeat of QuickTime" by Guillermo Ortiz in *develop* Issue 12.•

GRAFPORT BOTTLENECKS

QuickDraw provides some familiar examples of using bottlenecks. A grafPort contains pointers to all the low-level routines that it uses to implement its higher-level calls. By default the bottlenecks contain routines for drawing to the screen. When you create a grafPort, it's possible to swap out those ProcPtrs and put in your own. The default QuickDraw bottlenecks are usually swapped out in two circumstances: printing and getting information. Since all of QuickDraw must route through bottlenecks in the grafPort, and there are only 20 bottlenecks, a savvy Macintosh programmer will know which high-level routines call which low-level routine.

QuickTime introduces a new bottleneck — StdPix — to handle compressed image data. StdPix replaces the newProc1 bottleneck (see Chapter 4, "Color QuickDraw," of *Inside Macintosh* Volume V for details). You can sit in this bottleneck (that is, replace it with one of your own) and look at compressed data before it's decompressed.

Let's look at a situation where you may want to do this. The Picture Utilities Package is useful for getting information about pictures; however, it wasn't designed to support QuickTime. For example, GetPictInfo returns an inaccurate depth for QuickTime compressed images. The following code shows how to work around this problem. You replace all a grafPort's bottlenecks with dummy routines (so that nothing is actually drawn), except you can call GetCompressedPixMapInfo in the StdPix bottleneck. GetCompressedPixMapInfo returns the ImageDescriptionHandle for the picture, from which you can get the depth. DrawPicture eventually calls StdPix, among other bottleneck routines. Because the other bottlenecks were replaced with dummy routines, DrawPicture's behavior is reduced to just a StdPix call. The parameters passed to the StdPix routine fill out the parameters of the GetCompressedPixMapInfo routine, which in turn retrieves the pixel depth via the ImageDescription structure. The sample code on the CD creates a window for this function to "draw" in.

```
short gDepth = -1;

pascal void myStdPix(PixMapPtr src, Rect *srcRect,
   MatrixRecordPtr matrix, short mode, RgnHandle mask, PixMapPtr matte,
   Rect *matteRect, short flags)
{
   ImageDescriptionHandle     desc;
   Ptr                        data;
   long                       bufferSize;

   GetCompressedPixMapInfo(src, &desc, &data, &bufferSize, nil, nil);
   gDepth = (**desc).depth;
}
```

```
pascal void myTextProc(short byteCount, Ptr textBuf, Point numer,
                       Point denom){}
pascal void myLineProc(Point newPt){}
pascal void myRectProc(GrafVerb verb, Rect *r){}
pascal void myRRectProc(GrafVerb verb, Rect *r, short ovalWidth,
                       short ovalHeight){}
pascal void myOvalProc(GrafVerb verb, Rect *r){}
pascal void myArcProc(GrafVerb verb, Rect *r, short startAngle,
                       short arcAngle){}
pascal void myPolyProc(GrafVerb verb, PolyHandle poly){}
pascal void myRgnProc(GrafVerb verb, RgnHandle rgn){}
pascal void myBitsProc(BitMap *bitPtr, Rect *srcRect, Rect *dstRect,
                       short mode, RgnHandle maskRgn){}


void GetQTImagePixelDepth(PicHandle picture)
{
   CQDProcs bottlenecks;

   SetStdCProcs(&bottlenecks);            // Define our own bottlenecks.
   bottlenecks.textProc = (Ptr)myTextProc;
   bottlenecks.lineProc = (Ptr)myLineProc;
   bottlenecks.rectProc = (Ptr)myRectProc;
   bottlenecks.rRectProc = (Ptr)myRRectProc;
   bottlenecks.ovalProc = (Ptr)myOvalProc;
   bottlenecks.arcProc = (Ptr)myArcProc;
   bottlenecks.polyProc = (Ptr)myPolyProc;
   bottlenecks.rgnProc = (Ptr)myRgnProc;
   bottlenecks.bitsProc = (Ptr)myBitsProc;
   bottlenecks.newProc1 = (Ptr)myStdPix;  // pixProc.

   // Install our custom bottlenecks to intercept any compressed
   // images.
   (*(qd.thePort)).grafProcs = (QDProcs *)&bottlenecks;
   DrawPicture(picture, &((**picture).picFrame));

   (*(qd.thePort)).grafProcs = 0L;  // Switch back to the default procs.
}
```

COMPONENT BOTTLENECKS

A QuickTime routine may be implemented by a component. In this case, the concept of sitting in bottlenecks applies in a useful way to QuickTime components. As you know, the Component Manager sends the routine selector to the component, and the component parses the selector in its main function. Since all the selectors flow through the main function, it would be extremely valuable to replace the component

with your own delegating component in order to watch the selectors flow by. Just as you can sit in a bottleneck and capture routines, you can capture a component, perform an operation, and delegate the rest to the captured component. Then you could identify the sequence of routines needed to implement specific functionality.

Fortunately, some components have standardized interfaces as defined by Apple. These public APIs make it easy to match up the selector to the routine name, as defined in the interface files. With the introduction of QuickTime 1.5, the API for the base media handler has been made available as defined in the file MediaHandlers.h.

With a delegating component, you could theoretically modify the behavior of any component. But whether you can modify a given component depends on whether it implements the target request. Many components in QuickTime don't implement this functionality, which is unfortunate. However, with the introduction of QuickTime 1.5, the media handlers support the target request. By allowing media handlers to be delegated, QuickTime 1.5 greatly opens its architecture, giving enhanced meaning to *multi*media. For example, the text media handler delegates the generic media handler and uses its media scheduling and editing functions to do all the hard work. If you want to write your own media handler, delegating the generic media handler is just what you need.

To create a generic delegating component, I'll use a sample supplied with the article "Techniques for Writing and Debugging Components" in *develop* Issue 12. The sample is called NuMathComponent. It's a simple matter to convert it into a generic delegating component.

1. Using a resource editor, replace the componentType, componentSubType, and componentManufacturer of the NuMathComponent.π.rsrc 'thng' resource with 'mhlr', 'vide', and 'angs', respectively. Using 'angs' for the manufacturer puts the component before 'appl' alphabetically. Because the Component Manager searches alphabetically, when a search is done by QuickTime for a component of type 'mhlr' and subtype 'vide', it grabs your component. This technique forces QuickTime to use your component, which then captures Apple's component.
2. Open the NuMathComponent.π project and open the NuMathComponent.c file.
3. Be sure to declare the **globals** variable at the top of the main function as

   ```
   PrivateGlobals**  globals = (PrivateGlobals**)storage;
   ```

   This declaration gives you access to the fields in your global storage.

QuickTime components that implement the target request include Apple generic media handler, Apple standard media handler, Apple video media handler, Apple sound media handler, Apple text media handler, movie controller, movie grabber video channel, and movie grabber sound channel.•

4. Delete the second switch statement in the main function and replace it with

```
if (globals)
   DelegateComponentCall(params,
         (**globals).delegateComponentInstance);
else
   result = paramErr;
```

5. In _NuMathOpen and _NuMathRegister, change the described component's componentType and componentSubType fields to 'mhlr' and 'vide', respectively.
6. Build the code resource for the generic capture component (the code from *develop* Issue 12 on the CD has all the necessary files). You'll have to turn the declaration of ComponentSetTarget into a comment if you're using QuickTime 1.5.

Your main function should look like the following sample code. Focus on the call to DelegateComponentCall, as it's the major change needed to make the generic delegating component. To use the delegating component, either put it in the System Folder and reboot or drag and drop it on Reinstaller II.

```
pascal ComponentResult main(ComponentParameters *params, Handle storage)
{
   // This routine is the main dispatcher for the NuMath component.
   ComponentResult   result = noErr;
   PrivateGlobals**  globals = (PrivateGlobals**)storage;

   // Did we get a Component Manager request code (< 0)?
   if (params->what < 0)
   {
      switch (params->what)
      {
         case kComponentOpenSelect:       // Open request.
            result = CallComponentFunctionWithStorage (storage, params,
               (ComponentFunction) _NuMathOpen);
            break;
         case kComponentCloseSelect:      // Close request.
            result = CallComponentFunctionWithStorage (storage, params,
               (ComponentFunction) _NuMathClose);
            break;
         case kComponentCanDoSelect:      // Can Do request.
            result = CallComponentFunction (params,
               (ComponentFunction) _NuMathCanDo);
            break;
```

```
            case kComponentVersionSelect:    // Version request.
               result = CallComponentFunction (params,
                  (ComponentFunction) _NuMathVersion);
               break;
            case kComponentRegisterSelect:   // Register request.
               result = CallComponentFunction (params,
                  (ComponentFunction) _NuMathRegister);
               break;
            case kComponentTargetSelect:     // Target request unsupported.
            default:                         // Unknown request.
               result = paramErr;
               break;
         }
      }
      else        // Was it one of our request codes?
      {
         if (globals)
            DelegateComponentCall(params,
               (**globals).delegateComponentInstance);
         else
            result = paramErr;
      }
      return (result);
}
```

Now let's go back to the old example: Open MoviePlayer, set the break on DelegateComponentCall, and anchor a Memory window at "Δ(sp+4)^+2" for TMON Pro or "show '(sp+4)^+2' l" for MacsBug. This displays the selector from the ComponentParameters data structure passed into DelegateComponentCall. You'll be able to read the selectors for the routines as they're passed into the main function of the component. Remember, you can compare these numbers with the interface files (there are no interface files for the video media handler because it doesn't have a public API). In TMON Pro, you can open a View window of the interface file and look at the selectors without leaving the debugger.

You can try other situations and other traps to see whether they call the video media handler. Or set breaks in the open, close, version, and register routines — to find out how Things! works, for example. If you bring up the Things! control panel and select your media handler, you'll see Things! calls a trio of routines — open, version, and close. Also, you can see what calls are made to the component on startup.

A simpler technique can be used if you just want to analyze the selectors. Enter MacsBug and execute **thing**, which will list the entry point for each component. Set a breakpoint on an entry point. You can now use the same "show" instruction to display the selector. If it uses a fast dispatch mechanism, the selector will be in the low-order

When you're exploring, it's useful to use the **dx** command to turn the Debugger and DebugStr traps on and off. In TMON Pro, you can use the Options window to achieve the same result. If you set debugger traps in all the component requests, you'll inevitably be annoyed by the constant breaking.•

word of register D0. To modify this sample to be a media handler, you need to keep the same basic structure but support some or all of the selectors defined in the MediaHandlers.h file. For a description of those routines, refer to the *QuickTime Version 1.5 for Developers* CD.

OLD WORLD MEETS NEW WORLD

QuickTime routines can be intercepted and specially processed at various locations. Debuggers interact with QuickTime routines via the A-trap mechanism, providing valuable information about the sequence of routines needed to implement specific functionality. Applications can interact with QuickTime routines at the component level, allowing the program to change the routine's behavior.

The themes presented in this article extend beyond QuickTime. When newer technology comes from Apple, you can apply the common Macintosh themes of bottlenecks, contexts, and breaking on A-traps to new managers. Understanding these themes and applying them expedites your learning dramatically. In addition, you're now armed with techniques for investigating future Macintosh managers, some of which will be implemented through use of components. The techniques discussed in this article can help you flatten your learning curve, which can only be an advantage.

RECOMMENDED READING

- "Techniques for Writing and Debugging Components" by Gary Woodcock and Casey King, *develop* Issue 12.
- "Time Bases: The Heartbeat of QuickTime" by Guillermo A. Ortiz, *develop* Issue 12.
- "QuickTime 1.0: 'You Oughta Be in Pictures'" by Guillermo A. Ortiz, *develop* Issue 7.
- TMON Professional *Reference Manual* and *Tutorial* (Icom Simulations, Inc.).
- *QuickTime Developer's Guide*, available from APDA as part of the QuickTime Developer's Kit (#R0147LL/B), and the System 7.1 documentation. These have information on the Component Manager.
- *Inside Macintosh* Volume V (Addison-Wesley, 1986), Chapter 4, "Color QuickDraw."

THANKS TO OUR TECHNICAL REVIEWERS
Jim Batson, Peter Hoddie, Guillermo Ortiz, John Wang, Gary Woodcock•

PRINT HINTS

LOOKING AHEAD TO QUICKDRAW GX

PETE ("LUKE") ALEXANDER

With the release of QuickDraw GX later this year, there are a few changes in store for print land. In this column, I'm going to talk about two of these changes — the disappearance of the PDEF 10 resource from the QuickDraw GX LaserWriter driver, and the disappearance of the 'STR ' (-8192) and 'PAPA' (-8192) resources from a system running QuickDraw GX.

Like most changes, the fact that these resources are going away is both good news and bad news. The good news is that printing is going to work much better in the GX world than it does today. For one thing, the new printer driver architecture provides functionality that used to be unavailable through the Printing Manager, which is why people got involved in directly grabbing resources in the first place. The bad news is that if you *are* currently grabbing these resources, you're going to have some problems running under QuickDraw GX, and you need to decide now how you're going to deal with this.

Before we get into solutions, however, let's back up a bit, and answer the question, What's a PDEF, anyway? A PDEF is just a code resource for printer drivers. All printer drivers contain multiple PDEFs, each of which implements a piece of the driver's functionality, like displaying dialog boxes or alerts. PDEF 10 contains the printer access protocol (PAP) code that enables a LaserWriter driver to communicate with a network printer. (For details about the interface to PAP, see Chapter 10, "Printer Access Protocol," in *Inside AppleTalk*.) Some applications that need the PAP code have acquired it by sucking PDEF 10 out of the LaserWriter driver. Before Apple put together its PAP software licensing package, grabbing PDEF 10 like this was a quick and easy way to get what you needed. But it's an approach that's unsupported by Apple and that has always carried the seeds of compatibility problems.

So what should you do if you're currently grabbing PDEF 10 out of the LaserWriter driver? You've got the following possible solutions:

- Continue to grab PDEF 10, but make sure you're prepared to handle failure gracefully when you can't get it. Basically, you'll need to let the user know that your application isn't compatible with the version of system software being used — that is, the QuickDraw GX–based system. Not a very user-friendly solution, but that might be OK for your application.
- If all you need to do is download a PostScript™ file to the LaserWriter, use the PostScriptHandle PicComment and a basic print loop. Most applications need to do a lot more than send a bunch of PostScript code down the pipe — for instance, they need to set the characteristics of the PostScript printer — but if your application's needs are really this limited, it can be happy in the GX world. If you're interested in this approach, take a look at the Technical Notes "A Printing Loop That Cares . . ." and "PicComments — The Real Deal." You can also look at the PostScriptHandleDemo snippet in the Snippets folder on the *Developer CD Series* disc.
- Write your own PAP interface files. This is a relatively time-consuming operation, but it gives you total ownership of the code, and your application is unlikely to break with future system software releases. Quite a few developers have had success with this approach.
- License Apple's PAP library code — the same library that's used within the LaserWriter Font Utility. The latest version (v. 1.5) of the library improves on the last version and has more complete documentation.

PETE ("LUKE") ALEXANDER In the winter, Luke likes to hit the cross-country ski tracks. There you'll find him striding across meadows and down hills on long strips of fiberglass. One day he agreed to a "small" uphill ski at a friend's urging, only to discover halfway into it a virtually vertical two-mile climb! Weighing the risks of descending uncontrollably from there versus continuing on, Luke forged ahead, arriving at the summit an hour (and a lot of side stepping) later. Since then, he's been particularly wary of e-mails beginning "Dear Luke, I have this *small* printing problem."•

Grabbing the PAP driver from PDEF 10 is described both in the Macintosh Technical Note "Printer Access Protocol Q&As" under "Using PAP & code for finding printer driver under System 7" and in the article by Mike Schuster called "Laser Print DA for PostScript" in *MacTutor* Volume 2, Number 2.•

If you're interested in licensing this library, please contact

Software Licensing
Apple Computer, Inc.
20525 Mariani Avenue, M/S 38-I
Cupertino, CA 95014
AppleLink: SW.LICENSE
Phone: (408)974-4667

- Use Apple's LaserWriter Font Utility and add your custom features to it. In System 7, the LaserWriter Font Utility can handle drop-in enhancements called UTILs. UTILs take care of all the communication issues for you, and they give you quite a bit of power and flexibility. If you're interested in this approach, take a look at "PostScript Enhancements for the LaserWriter Font Utility" in Issue 10 of *develop*.

At this point, you're probably wondering which solution is the best for you. It all depends on your requirements. From my point of view, any solution besides the first is acceptable if it will keep your users happy and keep you compatible with future system software releases.

Now let's turn to the disappearance of the 'STR ' (-8192) and 'PAPA' (-8192) resources. On a non-QuickDraw GX system, the 'STR ' resource contains the name of the currently chosen printer driver, while the 'PAPA' resource contains the name and network address of the current PAP printer. Because QuickDraw GX will allow more than one active printer at a time, these resources will become obsolete.

Why is this a problem? Only because some applications currently grab the 'STR ' and 'PAPA' resources to automatically select a printer without going through the Chooser. Under QuickDraw GX, it will be possible to redirect a print job to another printer via the Printing Manager's public API. So new applications and applications revised to work under QuickDraw GX will be able to programmatically redirect print jobs on the fly in a clean, supported manner.

If your application isn't GX-smart, the simplest way to deal with the disappearance of the 'STR ' and 'PAPA' resources is probably just to remove your application's feature of circumventing the Chooser. If your application absolutely requires this functionality, however, make sure that when these resources aren't available, you tell the user to go use the Chooser. This way your application will print on a QuickDraw GX system without any problems.

This column has looked at a couple of compatibility problems that can emerge when non-QuickDraw GX applications run under QuickDraw GX, and also at some of the ways that you can avoid these problems. As long as you're prepared to implement one of the solutions recommended here, your application will run just fine under QuickDraw GX. And on the bright side, QuickDraw GX is going to allow your application to access and control a vast amount of information at print time. QuickDraw GX will provide the API your application requires to take care of all the user's printing needs without having to cruise through printer driver or system resources. You asked for it, and soon you'll have it!

REFERENCES

- *Inside AppleTalk* (Addison-Wesley, 1990), Chapter 10, "Printer Access Protocol."
- Macintosh Technical Notes "PicComments — The Real Deal" (formerly #91), "A Printing Loop That Cares . . . " (formerly #161), and "Printer Access Protocol Q&As" under "Using PAP & code for finding printer driver under System 7."
- "PostScript Enhancements for the LaserWriter Font Utility" by Bryan K. Ressler, *develop* Issue 10.

To find out whether the QuickDraw GX Printing Manager is installed, call the Gestalt function with the 'pmgr' selector.•

Thanks to Hugo Ayala, Tom Dowdy, Dave Hersey, Jim Luther, and Scott ("Zz") Zimmerman for reviewing this column.•

MACINTOSH DEBUGGING: THE BELLY OF THE BEAST REVISITED

This is a supplement to the article "Macintosh Debugging: A Weird Journey Into the Belly of the Beast" in Issue 8 of develop. *It presents a few debugging tools that were discussed at Apple's Worldwide Developers Conference in May 1992. Like those discussed in the previous article, these tools are designed to help you force the nasty, subtle bugs in your code to show their hideous little faces immediately, rather than lying in wait and biting you when you least expect it.*

FRED HUXHAM AND GREG MARRIOTT

ADAPTED FROM THEIR TALK AT THE WWDC BY DAVE JOHNSON

People often ask us, "How can I be a totally awesome, godlike debugging stud [or studette] like you?" Unfortunately, the big truth from the Issue 8 debugging article is just as true now as it was then: *debugging is hard.* That's just the way it is. The only way to get better at it is to practice. Now that we've got that straight and before we get into describing the new debugging tools, here are three pearls of wisdom to guide you in your practice.

First of all, it helps to know a lot about the operating system. The better sense you have of how the Macintosh works, the better off you'll be trying to track down a nasty bug. Dare to delve into the bowels of the OS. Read and reread *Inside Macintosh*; take it with you to bed, to the bathroom, out to dinner, and on dates. (You might want to invest in a sturdy wheelbarrow, especially with the new *Inside Macintosh* volumes proliferating like rabbits.) For that matter, read every Macintosh programming book ever written (especially those listed at the end of this article) and every Technical Note, Snippet, piece of Sample Code, and issue of *develop*, as well as every word on the AppleLink Discussion boards. Also, spend lots of time in debuggers, watching the system do its thing. If you're not dreaming in hex, you're not spending enough time in MacsBug.

Second, get slammed a lot. The people who are the best at debugging are usually the ones who've had to track down the most bugs and therefore have an encyclopedic knowledge of them. If you have a really nasty bug in your code that crashes the machine on a seemingly random basis and takes you three days to find and squash,

FRED HUXHAM (AppleLink FRED) was born and raised in California. He used to be a tremendous athlete, know bazillions of babes, and go to wild parties in New York and California with people like Andy Warhol and Keith Haring. Now he's 15 pounds heavier, knows only one babe (his wife), and thinks a day spent sitting on his roof deck watching boats go through the Golden Gate is really exciting.•

GREG MARRIOTT (AppleLink GREG) is a SWM, 28, 6'0", 195 lbs., brown hair and eyes, sincere, hardworking, good sense of humor. Enjoys music, romantic walks, quiet evenings, and good books. Seeks nice woman for friendship and more. Send photo.•

then by jove you'll remember that bug the next time you see it. Simply put, the more bugs you find, the better you'll be at finding bugs.

Last, use good tools, and use them all. Reread the Issue 8 article. Turn on those tools and stress your code. Bend, fold, staple, and mutilate it. Show no mercy.

These things will help you on your way to becoming a primo bug stomper, but debugging is like any complex skill in that it can't really be taught past a certain point. You simply have to do it a lot, and over time you'll get better. Tools and techniques such as the ones presented here can help enormously, especially by forcing hidden bugs to the surface, but they can never do the whole job for you.

This time there are only four new tools to talk about — Double Trouble, Dispose Resource, Blat, and Smart Friends — so this article is much shorter than the last one. The tools are available on the *Developer CD Series* disc, as well as on AppleLink and elsewhere. We're doing this backward from the last time: first we'll present a buggy code sample, then we'll talk about the tool that would find the bug.

DOUBLE TROUBLE

Can you find the bug in this code sample?

```
myHandle = NewHandle(100);
if (myHandle) {
   AddResource(myHandle, 'dumb', 10, "\p");
   if (resError()) HandleTheError();
   CloseResFile(outputFileRef);
   DisposeHandle(myHandle);
}
```

OK, time's up. This one's not too hard. The problem is that during CloseResFile the Resource Manager disposes of all the resources in memory. The DisposeHandle call afterward is unnecessary and is actually potentially disastrous. Normally you'll just get an error and DisposeHandle will do nothing, but occasionally the data structures in the Memory Manager will conspire to really screw you.

Here's how: Master pointers are allocated in clumps called master pointer blocks, which are nonrelocatable blocks in your application's heap. The master pointers that are currently free for use are kept in a linked list by the Memory Manager. The list is LIFO, like a stack: when you allocate a new handle, the Memory Manager uses the first master pointer in the free list, and when you dispose of a handle the freed master pointer is returned to the beginning of the list.

Now the plot thickens. If the first master pointer in the free list also happens to be the first master pointer in its master pointer block (so that the master pointer and the

master pointer block have the same address) and then you dispose of a handle twice by mistake, *very bad things* will happen. On the first dispose, everything is fine: the Memory Manager frees the block the master pointer points to and returns the master pointer to the start of the free list. At this time, the master pointer *still* points to a valid block of memory, but now it's the master pointer block itself! So on the second, unintentional dispose, when the Memory Manager dutifully frees the block for reuse, you're set up for disaster. Subsequent memory use will likely result in writing over many master pointers, which will of course trash you one way or another.

Figure 1 illustrates this scenario. On the left is the top part of a master pointer block that resides in the heap at address 80. The heap's free list is a standard linked list (each entry contains the next entry's address) beginning at hFstFree. Note that the first entry in the heap's free list is also the first master pointer in the block. This is the first step to trouble.

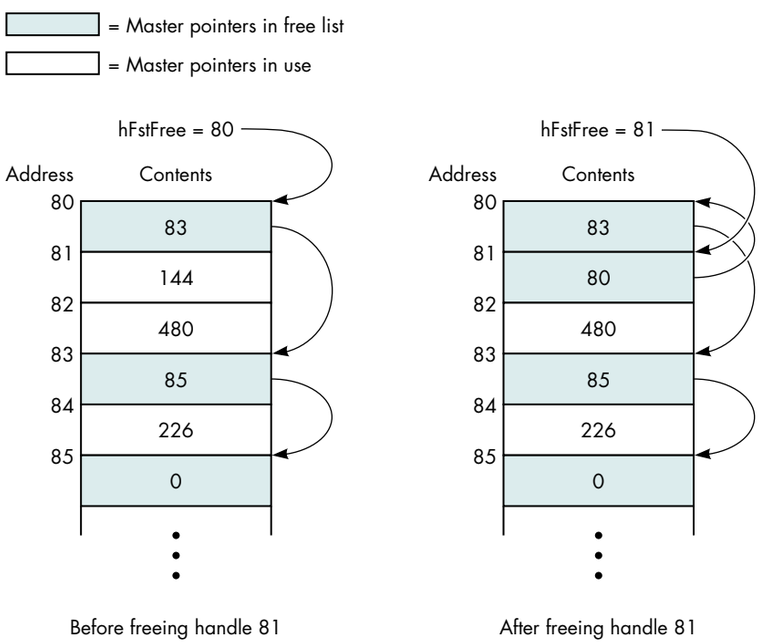


Figure 1
How Disposing of the Same Handle Twice Can Spell Disaster

Now we call DisposeHandle on the master pointer at 81. DisposeHandle looks at the block pointed to by the master pointer (in this case the block at 144, not shown), determines that it is indeed a valid block, marks it as free for reuse, and adds the

newly freed master pointer to the front of the free list. So far so good. Now the master pointer block looks like the one on the right in the figure.

Then we call DisposeHandle on 81 again by mistake. DisposeHandle looks at the block pointed to by the master pointer (now it's the block at 80, our master pointer block!), determines that it is indeed a valid block (uh oh), marks it as free for reuse (yikes!), and adds the newly freed master pointer to the front of the free list — and the heap is now hosed for good. This Memory Manager bug is subtle and rare, but oh so nasty.

Even if you're lucky enough to avoid this particular sequence of events, a double disposal is definitely a bug. Double Trouble is a system extension that watches calls to DisposeHandle to make sure it's not being called on something in the free list. If it is, Double Trouble drops into the debugger with a suitable warning.

We'll be the first to admit that Double Trouble is far from perfect. It infers the existence of heap zones by watching InitZone and then trying to figure out when a heap isn't a heap anymore. The possibility exists that it will guess wrong and cause a bus error when trying to walk a free list that's no longer a free list. Furthermore, in some cases Double Trouble can noticeably slow down parts of the system. (After playing a long QuickTime movie, for instance, the machine may freeze for almost a minute.)

But despite Double Trouble's shortcomings, we do still recommend running it all the time. Just try to remember that it's running so you don't chase your tail trying to find the cause of occasional mysterious slowdowns.

DISPOSE RESOURCE

Here's the code. What's the bug?

```
myPicture = GetPicture(kPicID);
if (myPicture) {
   DrawPicture(myPicture, &myRect);
   DisposeHandle(myPicture);
}
```

That's right, you should never call DisposeHandle on a resource handle. If you do, the Memory Manager will free it just fine, but the Resource Manager has another reference to it, stored in the resource map, that will be left dangling. Later on, since the Resource Manager doesn't know the handle was disposed of, it may try some manipulation with the handle. The results may not crash you immediately, or at all — it depends on what the operation is and what's in the handle — but they're certainly not what was intended. Instead of DisposeHandle, you should always call ReleaseResource on resource handles. ReleaseResource will properly dispose of the

handle *and* will update the resource map. (Note that KillPicture won't do the right thing here either; it's intended for pictures created via OpenPicture, not for PICT resources.)

Dispose Resource is another extension a lot like Double Trouble. It also watches DisposeHandle calls, this time looking to see if the handle being disposed of is a resource handle. If so, you'll drop into the debugger with a suitable warning.

Dispose Resource has one idiosyncrasy you should know about: it's been known to indicate "false positives." Some parts of the system (we haven't been able to track down which ones yet) seem to save a resource handle's state, detach the resource, and then restore the state of the handle (restoring the resource bit!). Use Dispose Resource. It will ensure that you don't make the same mistake.

BLAT

This time the code's in assembler:

```
; Offset the rect by 128 pixels in each direction.
PEA      theRect(A6)
MOVE.W   $0080, -(SP)
MOVE.W   $0080, -(SP)
_OffsetRect
```

If you have "iron man" syndrome and insist on programming in assembly language, this can happen to you. We forgot to type a # in front of each $0080. As a result, instead of moving the number $0080 (128) onto the stack twice in preparation for the OffsetRect call, we're moving the contents of memory location $0080. Often this kind of bug is immediately obvious, but not always. If you're moving a Boolean, for instance, you have a fifty-fifty chance of getting the right value, even though you're getting it from some random spot in memory. It's those cases that will give you debugging headaches.

One easy (and recommended) way to avoid the problem in this example is to write in a higher-level language. But we realize that's not always possible, and besides, this is really a whole *class* of problems: reads and writes from places in memory you didn't intend. The best way to catch this wild memory reference kind of problem is, naturally, with memory protection, something that — sadly — the Macintosh normally lacks. In the last article we mentioned Jasik's implementation, but now there's something else you should know about. Bo3b Johnson has written a dcmd called Blat that uses the MMU to protect memory locations 0–255 from both reads and writes.

Blat has been tested and works well on the Macintosh IIfx, IIx, and SE/30. Because its operation is so hardware dependent, it's hard to predict whether it will work on a

In Bo3b's name, the "3" is silent.•

given machine. Some basic guidelines are that it requires an MMU and won't work with 68040 machines or with most configurations of machines with the IIci ROM (IIci, IIsi, LC). For further details, see the release notes and the source code, thoughtfully provided by Bo3b along with the dcmd itself.

SMART FRIENDS

This bug is subtle, so pay close attention:

```
#pragma parameter __d0 GetA0
Ptr GetA0(void) = {0x2008};         // MOVE.l  A0,D0

void MyCompletionRoutine()
{
   long        saveA5;
   HooHahPtr   myHooHah;

   myHooHah = (HooHahPtr)GetA0();
   saveA5 = SetA5(myHooHah->myA5);
   gSomething[0].flag = true;    // Set a flag in a global array.
   SetA5(saveA5);
}
```

This code really tries hard to do everything right. As the name implies, it's a completion routine, so it could be called at interrupt time. First a pointer to the data is retrieved from A0, and then A5 is set to a previously saved value, thus allowing the routine to access its global variables. Once A5 is set up, the global reference can be made safely. Finally, A5 is restored to its previous value to clean up. Sounds great, right? The only problem is, it doesn't work.

Here's why: the MPW C compiler will actually set up the global reference *before* the SetA5 call, so accessing the global accesses some unknown part of memory. This is legal compiler optimization behavior! If GetA0 and SetA5 were functions or traps, the bug would disappear, but since they're declared inline the compiler doesn't feel compelled to delay the evaluation of the global array reference. The solution is to set up A5, then call a different routine that does the global reference.

Now in this case, how do you think we — the debugging gods — figured out the bug? We tried the first few things we could think of; but then when we weren't making headway after a few probes, we didn't just sit there and suffer in silence, banging our heads against the proverbial wall. We called in some Smart Friends! The veil of illusion was torn from our eyes, and we were shown the heart of the truth (in other words, one of them had seen this bug before). The point is that in debugging, two (or more) heads are far, far better than one. Bugs are not like germs: when you share them, everyone benefits. Maybe your very own Smart Friends have had a

similar bug before, so they'll recognize immediately what's going on. Or maybe they'll think of something different to try. At the very least, they'll temporarily divert you from your frustration, maybe make you feel less stupid, and then you can all go out for pie together.

THAT'S IT!

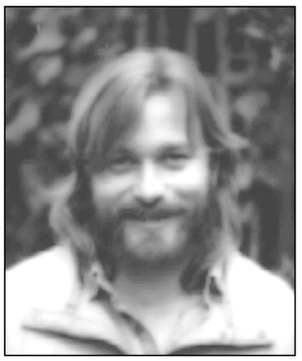
Add these tools to your arsenal of bug sprays and foggers. Use them all and use them well, and you, your code, and your customers will be far better off.

FURTHER READING

Bedside books for the serious student of debugging:

- *How to Write Macintosh Software,* 3rd ed., by Scott Knaster and Keith Rollin (Addison-Wesley, 1992).
- *Macintosh Programming Secrets,* 2nd ed., by Scott Knaster and Keith Rollin (Addison-Wesley, 1992).
- *Debugging Macintosh Software with MacsBug* by Konstantin Othmer and Jim Straus (Addison-Wesley, 1991).
- *MC68000 Family Programmer's Reference Manual* (Motorola, Inc.).

THANKS TO OUR TECHNICAL REVIEWERS
Jim Reekes and Bryan Stearns•



THE VETERAN NEOPHYTE

TOWER OF BABBLE

DAVE JOHNSON

I recently started learning MacApp. (I know, I know, I can see you shaking your great shaggy collective head, chuckling to yourself, asking where I was three years ago when MacApp was still news. Let's just say I'm a late bloomer.) People weren't kidding when they said that the learning curve is long and steep. They also weren't kidding when they said that it's absolutely worth it.

For me, it was a double whammy: learning MacApp *and* transitioning from THINK C to MPW. (See, if I'd only learned it in the MacApp 2.0 days I could have used THINK Pascal, but noooo, I had to wait till now.) I've been using THINK C for virtually all my programming since 1986 or so. Using MPW for my own little exploratory projects would be like calling in a highly trained, ultramodern, rapid-deployment mobile emergency medical team to remove a splinter from my thumb. The job would get done, and beautifully, but it'd be an absolutely colossal waste of time, effort, and expense. Frankly, I'd rather just have a good pair of tweezers.

But alas, if I want to use MacApp (and I do!) the days of coding on my PowerBook 100 in the backyard with loyal hounds lolling at my feet are gone for good. Now I need 16 MB of RAM minimum and at *least* 40 MB of hard disk space (120 to be really comfortable). And I'm not even going to *mention* MacApp compile times; it hurts me too much.

But all that's really just logistics and can be gotten used to pretty quickly. The real difference is in the very nature of my interaction with the machine: It used to be that when I'd think of something that needed doing, I'd just go do it. It was like building a machine from scratch, piece by handcrafted piece. Now, using MacApp, when I think of something that needs doing I conduct massive, cross-referenced searches through megabytes of source code to figure out where it's already been done, because no doubt somebody already thought of it, or something very much like it, and implemented it better than I ever could. It's as though I'm running around on top of a giant, humming machine that stretches to the horizon on all sides, hunting for just the right place to reach down into the dark recesses, pull up a live, vibrating cable, and splice in my little special-purpose unit. Often I've spent an hour hunting around for the right place to insert some code, only to discover that to do what I want I just need to set the value of some out-of-the-way Boolean deep inside an object's remote ancestor.

Well, I could ramble forever about my learning experiences, but those of you who've been there know all about it, and those of you who haven't probably don't want to hear it. But this is the first time since I discovered the Macintosh and switched from FORTH to C that the *feel* of programming has been completely transformed for me. It occurred to me that the fact that programming is the kind of thing that can *have* a feeling to it is noteworthy.

Programming computers is an activity unlike any other. It's a human-machine interaction, but because the machine is very special, interacting with it is also very special. Programming has a depth that other machine interactions don't, so it can assume qualities not normally associated with the operation of machinery. It can be a creative act, akin to building an intricate, glittering crystal clockwork out of gossamer strands of pure thought; and it can also be formidable drudgery, a mountain of mind-numbing details, endless in their intricacy, interrelatedness, and total irrelevance to the real task at hand. These are not normally the kinds of

DAVE JOHNSON recently bought some Crash Dummies and peripheral equipment. These are little "action figures," modeled after real crash dummies, that fly apart in various ways upon impact. You can buy a car to crash them in, crash dummy pets (named Bumper and Hubcat), crash dummy babies in strollers or car seats, crash dummy pedestrians, and even a crash dummy torture chair with straps and clamps and cranks to pull the dummies apart more slowly, one limb at a time. Dave is convinced that if he preserves all the parts in their original packaging he can sell them for some huge amount of money in the future, or at least that's how he's justifying the expense.•

things you'd say about operating your dishwasher or toilet.

Computers are something truly new on earth. They're machines that can simulate any other machine; they're somehow potentially *every* machine in one. A well-known computer luminary put it this way:

> *It [the computer] is a medium that can dynamically simulate the details of any other medium, including media that cannot exist physically. It is not a tool, although it can act like many tools. It is the first metamedium, and as such it has degrees of freedom for representation and expression never before encountered and as yet barely investigated.*
>
> — Alan Kay, "Computer Software," *Scientific American*, September 1984.

Other machines are physical extensions of ourselves; they let us sense and manipulate our physical world with more power and flexibility than we can by ourselves. But they're just *physical* extensions. Computers, though, manipulate and embody abstractions and symbols; they operate on patterns of electrical activity, on imagination, on mindstuff. If you can imagine a machine or a medium in detail, you can program a computer to simulate it. So programming computers is much, much more than telling them what to do — it's telling them what to *be*.

Of course, all this philosophical and poetic mumbo-jumbo crashes to the ground when faced with reality. Try telling my friend Michele — who wrote an entire book on her Macintosh SE and just recently realized that she can use Standard File dialogs to navigate her hard disk — that her computer "has degrees of freedom for representation and expression never before encountered." Yeah, right. Admittedly, the computer is much more fluid-seeming to programmers than to users (someday, hopefully, a moot distinction), but there's still a large discrepancy between the promise and the realization. Computers still feel more like erector sets — lots of hard, inflexible little parts — than like clay.

Boiled down to its thick, syrupy essentials, computer programming is quite simply the creation and communication of detailed instructions. The creation is the really exciting part, and is (or should be) the main task. But the communication is what really defines the experience of programming; it's the part that has a *feel* to it.

All this touchy-feely talk smacks of natural language. Are programming languages really just another class of natural languages? Is that why programming can feel so rich? I found a great book that addressed this very question (among others): *The Cognitive Connection* by Howard Levine and Howard Rheingold.

Programming languages and natural languages do indeed have deep similarities, and share essential features found in any language. They're both sets of abstract symbols that have meaning only by mutual agreement between communicating parties. They're both open-ended: they have an underlying structure and system of rules that allow an infinite variety of correct sentences to be constructed. (Even more remarkably, any correct sentence can later be deciphered by anyone who knows the language, even though they've never seen that sentence before.)

Linguists say that a language has three parts: phonology, syntax, and semantics. Phonology is the way a language is turned into sounds, and is irrelevant to programming languages since they're never spoken. Syntax is the set of rules that specify how the parts of the language — words and phrases — are put together to form sentences. Programming languages obviously have strict and unforgiving syntax. But syntax by itself is an empty shell, telling us only whether a sentence is well formed, not what it means. That's the function of semantics.

Ah, sweet semantics! This is where the rubber meets the road, linguistically speaking, and where significant differences between natural languages and programming languages begin to appear. Howard and Howard illustrate one big semantic difference between natural languages and programming languages

by comparing their dictionaries. (Dictionaries are, in a sense, the embodiment of a language's semantics.)

Natural language dictionaries are written in natural languages, so the language must be rich and flexible enough to describe itself. When you look up an English word in Webster's, you get a definition written in English. This is only possible because words in natural languages can have more than one meaning.

Programming language dictionaries, on the other hand, are never written in a programming language. When you look up the definition of a Pascal word, the description is written in English (or Portuguese or Swahili or whatever), not Pascal or C++ or LISP. Unfortunately, the duplicity of meaning that allows a natural language to describe itself opens the door to paradox and self-contradiction, something programming languages can't tolerate.

But there's another, even more apparent semantic difference between programming languages and natural languages. As the Howards so aptly put it:

> *. . . although philosophers and linguists have struggled for centuries to give precise meaning to the word "meaning," you don't need a degree in either discipline to realize that what constitutes meaning for a programming language is dramatically different from what constitutes meaning for a natural language.*

Semantically, programming languages are only a sort of horribly stunted subset of natural languages, because the world they describe — the operations of computers — is only a sort of horribly stunted subset of the natural world. So "conversations" in a programming language aren't conversations at all; they're one-sided and one-dimensional commands whose conversational interest is on a par with the instructions on the back of a shampoo bottle: Lather, rinse, repeat.

We are, of course, in the infancy of our relationship with computers, still drooling and babbling experimentally most of the time. Look at MacApp: compared to other available methods of programming the Macintosh, it's astoundingly elegant and streamlined, but even MacApp's most vocal devotees don't want to stop there. Far from being the end product of the evolution of programming, MacApp is only one of the first teetering steps toward more natural and more fluent communication with computers.

A big question is whether our interactions with computers will *ever* be totally fluent, where fluency means the complete subsumption of syntax, so that we can go directly from meaning to expression with no conscious effort. Some people insist it will happen, that there's a future of instant, effortless communication with computers, a wide and crystal clear pathway between us and them, but somehow I can't buy it. I suspect that instead, computer communication will just get more and more like natural communication.

Fraught with misunderstanding and misinterpretation, blocked by its implicit awkwardnesses and incompleteness, human language is nevertheless rich beyond depth. Its infinite flexibility allows it to carry and contain the full spectrum of human thought and feeling, and provides a ground for endless creativity. Indeed, there is an intense joy to using language — *any* language — well. If we get only half as far with our computers as we have with our words, we'll have come a very long way indeed.

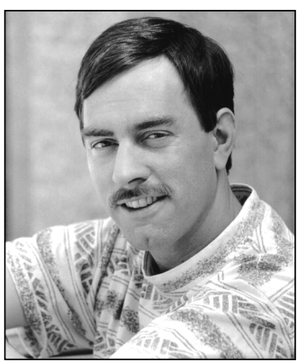
RECOMMENDED READING

- *The Cognitive Connection* by Howard Levine and Howard Rheingold (Prentice-Hall Press, 1987).
- *Scientific American,* September 1984.
- *The Happy Birthday Present* by Joan Heilbroner, pictures by Mary Chalmers (Harper & Row, 1962).

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

ADVENTURES IN COLOR PRINTING

Along with Color QuickDraw came the need for applications to support printing of pixMaps. Users need (and expect) to be able to produce realistic hard copies of their color screen displays. The challenge for developers is to ensure high-quality output regardless of the printing configuration being used. This article and its accompanying sample programs show you how.



DAVE HERSEY

Consider a 24-bit color image we've just scanned in. We'd like this image to print in color on all color printers, whether they're color LaserWriters, ImageWriters with color ribbons, or color ink jet printers. Similarly, we'd like to generate output that represents the source image as closely as possible when we're using grayscale printers such as the LaserWriter IIg with PhotoGrade, or monochrome printers such as LaserWriters without PhotoGrade, StyleWriters, and ImageWriters with black ribbons. And, of course, we'd like our images to look great even when the user has chosen black-and-white printing on a color-capable printer.

The challenge of producing high-quality output regardless of the printing configuration should ideally be handled at the driver level, through new printer drivers or solutions such as ColorSync or QuickDraw GX. But until every system makes use of these new technologies, we're stuck with the task of working around the pitfalls of the present printing architecture. The key is to determine the printing configuration we're working with and then supply the routine that ensures the highest-quality output in that particular case.

This article and the sample code that accompanies it on the *Developer CD Series* disc will show you how to print pixMaps (or pictures containing pixMaps) faithfully on any printer by building in a combination of approaches to cover all cases. The results will be far better than any you can get by a "one size fits all" approach. I'll discuss how to make use of Color QuickDraw when a printer driver can support it, how to render color images with original QuickDraw on printers whose drivers don't support Color QuickDraw (such as the ImageWriter), and how to convert color images to high-resolution halftone images for printing on monochrome printers.

DAVE HERSEY is a member of the Printing, Imaging, and Graphics (PIGs) group in Apple Developer Technical Support. Before leaving his boyhood home of Newport, Rhode Island, more than two years ago, Dave churned out code for a number of different software developers, writing applications that ranged from a popular accounting package to flatbed scanner software. When he's not absorbing radiation in front of his computer, Dave enjoys vacationing at the family summer camp in Wayne, Maine (no kidding), watching CNN on his 35-inch television ("It's still not big enough"), and playing Duplos with his nephews. Even with such a busy agenda, Dave still finds time to torment his peers with occasional practical (and impractical) jokes, in true DTS style.•

The methods in this article apply equally well to PostScript and QuickDraw printers, and they work correctly whether or not the new printing solutions are in place. Note, however, that without some extra work (see the end of this article) these methods may not be optimal for printing pictures that contain text. When text is converted to pixMaps, all of the font information is lost, and the result can often be chunky, poor-quality text that's hard to read.

All of the techniques described here require you to have 32-Bit QuickDraw available. This covers any Macintosh with 32-Bit QuickDraw in ROM and any machine with Color QuickDraw in ROM that either is running System 7 or has the 32-Bit QuickDraw INIT installed. If you have only Color QuickDraw available (the version that predates 32-Bit QuickDraw), you can still use all of the techniques described here as long as you implement a GWorld structure and replacements for the calls OpenCPicture, NewGWorld, DisposeGWorld, and CopyBits with ditherCopy mode. Methods to apply when Color QuickDraw is not available are discussed in "Making the Most of Color on 1-Bit Devices" in *develop* Issue 9. Together, the present article and the article in Issue 9 give you solutions that cover printing in any situation.

THE "ONE SIZE FITS ALL" APPROACH: A BAD IDEA

Many applications today that deal with pixMap images don't worry about addressing all the possible variations in printing configurations. This is unfortunate because the "one size fits all" approach can severely limit an application's potential.

Under the current printing architecture, if you provide just one printing method in your application based on assumptions about the printing configuration most likely to be used, you're bound to frustrate and annoy some users. For example, imagine a user with a color laser printer who for some special purpose wants to print a color image in black and white. If your application has failed to take this printing possibility into account, the user will end up with a hideous Black Blob that looks nothing like the original. Or picture a user with an ImageWriter who decides to invest in a color ribbon so that she can print color images with her favorite paint program, only to discover that because the program doesn't provide for this possibility, the result is — you guessed it — a hideous Black Blob. Or consider the users who find that documents containing color images that print just fine on their LaserWriters at work, print terribly on their StyleWriters, ImageWriters, or Personal LaserWriters at home. These frustrated users will end up clogging your customer service hotline with the kind of calls you don't want to get. The moral of the story is that under the current printing architecture it's not enough to provide just one method to print your images.

Far superior to the "one size fits all" approach is the strategy of providing printing routines to address the whole range of printing configurations your application might encounter. Then all your application has to do at print time is to determine which

printing configuration it's dealing with and provide the appropriate printing routine. That's what this article is about.

We start by looking over the possible printing configurations; then we consider routines to address each of these configurations; and finally, we look at how an application can determine which printing configuration it's facing.

THE POSSIBLE PRINTING CONFIGURATIONS

When you're printing from the Macintosh, there are three distinct types of printer drivers that you might encounter:

- Printer drivers that support Color QuickDraw calls. For example, the LaserWriter driver 6.0 and later in Color/Grayscale mode, printing to color, grayscale, or monochrome laser printers; and drivers for a number of third-party color laser printers, ink jet printers, film recorders, and so forth.
- Drivers for color-capable printers that don't support Color QuickDraw calls or data structures. For example, the ImageWriter drivers through version 7.0 printing to an ImageWriter with a color ribbon installed.
- Drivers for monochrome printers that don't support Color QuickDraw calls or data structures. For example, the ImageWriter drivers through version 7.0 printing to an ImageWriter with a black ribbon installed, the StyleWriter using the 7.2.2 driver, and laser printers using the LaserWriter driver 5.2 (or 6.0 and later in Black & White mode).

Note that what matters to you isn't the printer being used, but the printer driver. Thus, for example, if you print Color QuickDraw to a LaserWriter IINT using the version 5.2 driver (which doesn't have the Color/Grayscale option), you'll end up with nothing but stark black shapes because there's no Color QuickDraw support in the driver. The same printer using the 7.0 driver with the Color/Grayscale option selected will produce excellent results in response to the very same drawing commands — same printer, but totally different results depending on the driver. Another good example is the ImageWriter. Versions of the ImageWriter driver through version 7.0 don't support Color QuickDraw calls, but there are third-party drivers for the ImageWriter that do.

Note also that in the category of drivers that support Color QuickDraw calls, no distinction needs to be made between grayscale and color printers. Based on your experience with Color QuickDraw on the screen, you might have the impression that a color image should be converted to a grayscale image before printing to a noncolor device, or that you need to get the printer port's color table, GDevice, or bit depth, and map your images to those before printing. But in fact, this is not only

unnecessary but also undesirable in the printing environment. If the driver supports Color QuickDraw, you don't need to worry about whether your images will be printing on a color or a grayscale printer.

ABOUT PRINTER DRIVER PORTS AND COLOR QUICKDRAW SUPPORT

While I've categorized printer drivers by whether or not they support Color QuickDraw, what we're really concerned with is whether they give us a cGrafPort or a grafPort to draw in. The port I'm referring to here is the TPPrPort that the driver returns to the application through PrOpenDoc. Printer drivers that give us a cGrafPort support Color QuickDraw calls, because a cGrafPort is capable of handling multibit pixels. On the other hand, printer drivers that give us a grafPort don't support Color QuickDraw calls.

Drawing with Color QuickDraw in a grafPort, while possible, will yield disappointing results. Consider what happens if you try to CopyBits a 24-bit-deep image to the ImageWriter (assuming you're not using ditherCopy mode in System 7). Since you're copying to a driver port that's capable of only two colors, every one of the pixels in your image will become either your foreground color or your background color, whichever its value is closest to. In the usual case of a black foreground and a white background, you'll end up with the Black Blob effect — all colors with luminance values of at least 50% black draw black and everything else draws white.

Although the situation is improving, at present most of the drivers that Apple ships return grafPorts. (See "The Story Behind Color QuickDraw Support" for the whys and wherefores.) The LaserWriter drivers version 6.0 and later are capable of providing a cGrafPort for your application to draw into, but note that if the user selects Black & White mode in the color LaserWriter driver's print job dialog, even that driver returns a grafPort; a cGrafPort is returned only when the user has chosen Color/Grayscale mode.

Let me warn you up front that the printer driver port isn't necessarily a true cGrafPort or grafPort — that is, one that's valid outside the context of the Printing Manager. In the case of Apple's printer drivers, it never is. The fact is that drivers have a lot of leeway when it comes to the port structure they return. Since the driver needs to replace the port's QuickDraw bottleneck procedures in order to direct the data to a printer, there's no need for many of the fields that you would use if you were drawing to a true grafPort or cGrafPort, such as a window on the screen. In fact, when you make a call like

```
CopyBits(&bitMap, &printPort->gPort.portBits, &srcRect, &destRect,
   srcCopy, nil);
```

the data most likely won't even end up in the driver port's bitmap. In fact, the bitmap structure may not even exist. There's no need for it to. All that matters is that as you

THE STORY BEHIND COLOR QUICKDRAW SUPPORT

So why is it that the LaserWriter didn't support cGrafPorts until the 6.0 LaserWriter driver? And why is it that the 7.0 ImageWriter driver still doesn't support cGrafPort printing?

The first answer is simple. Color QuickDraw didn't exist when the LaserWriter driver was created back in 1985. It wasn't until 32-Bit QuickDraw came on the scene that the driver was revised to support color/grayscale printing. Since the driver wasn't originally designed with Color QuickDraw in mind, this support represented major changes to the source code. As such, it took until version 6.0.2 for most of the glitches to be worked out. Even today, the LaserWriter driver is essentially an old-style QuickDraw driver with Color QuickDraw support patched in.

The ImageWriter driver never was revised, except to add color tables to the print job dialogs in the 6.1 version. Why wasn't the driver revised? Well, for the ImageWriter driver to fully support Color QuickDraw, it would essentially need to be rewritten. Since there's been no overwhelming demand and since color printing solutions are available via the color LaserWriter driver and third-party printers and drivers, no one has rewritten the driver to provide color support.

At some point in the future, all of Apple's printer drivers will support Color QuickDraw calls. But for now, applications should be aware that a printer driver returns either a cGrafPort or a grafPort, and it's the application's responsibility to "do the right thing" regardless of the port type.

draw into the grafPort or cGrafPort, your drawing commands are intercepted, possibly translated, and then redirected to the printer.

So don't assume that the printer driver's port is a true grafPort or cGrafPort, or that the values therein have anything to do with how your image will print. You should view the printer driver's port as a private structure, with the only public fields being the actual pointer to the grafPort or cGrafPort (your TPPrPort pointer) and its port's portBits bitmap. Even then, SetPort and CopyBits are the only calls you should pass those values to.

THE PROBLEM AT HAND

To get back to the problem at hand, we need printing routines to address each of the three possible printing configurations. The rest of this article is devoted to describing those routines and outlining how to determine at print time which routine is appropriate. The routines are demonstrated by four samples in the Adventures in Color Printing folder on the *Developer CD Series* disc.

Note that all the samples implement the technique of loading and storing print records from job to job. All printing applications should implement some sort of handling like this so that when users attempt to print documents, their last used settings are available, rather than the driver's defaults.

All samples work under System 6 or 7. Remember that to use the methods described here, you must have 32-Bit QuickDraw available, or if you have only Color QuickDraw (the version that predates 32-Bit QuickDraw) available, you must implement a GWorld structure (which is the same thing as a cGrafPort) and replacements for the calls OpenCPicture, NewGWorld, DisposeGWorld, and CopyBits with ditherCopy mode.

PRINTING WITH COLOR QUICKDRAW SUPPORT

The easiest color printing situation you'll come across occurs when a printer driver gives you a cGrafPort to work in. To generate the best results we first need to deal with setting the resolution and scaling the image. Then we want to band our image through a 32-bit-deep GWorld to avoid the potential problem of operator incompatibility. The Color Adventures sample code demonstrates how we go about this. As mentioned earlier, grayscale printing in a cGrafPort shouldn't be treated any differently from color printing in a cGrafPort.

SETTING RESOLUTION AND SCALING THE IMAGE

When we print an image, a couple of different scaling operations are involved. First, our application sets the printer driver port's resolution and, if necessary, scales the image to that resolution; then the printer driver scales the image to the device's physical (output) resolution during printing. The amount an image is scaled when copied to the printer port is calculated as follows:

scaleAmt = (sourceDPI / destinationDPI) * (scaling factor from Page Setup dialog)

To achieve the highest-quality output, our image's resolution should ideally be the same as the printer's physical resolution. If our image's resolution doesn't match the printer's resolution, we can scale the image before printing, change the port's resolution to match the image resolution, or do a combination of both (scale the source image and the port).

Here's how we proceed: First, we need to know the resolution of our source image. Most PICT files on the Macintosh are rendered at 72 dpi, but that needn't be the case, and in the case of scanned images is actually rather unlikely. The GetImageRes routine in the Color Adventures sample shows how to find the resolution of any PICT. If the OpenCPicture call was used to create the picture, the resolution information is stored right in the picture header for easy retrieval. Otherwise, we need to determine the resolution by parsing the picture.

Once we have the image resolution, we need to know how close the printer can be set to that resolution. We can determine the supported resolutions for a particular printer using PrGeneral, as discussed in the article "Meet PrGeneral" in *develop* Issue 3 and in *Inside Macintosh* Volume V. As noted in those sources, when we call PrGeneral with the GetRslData opcode, drivers that support PrGeneral will return a

list of discrete resolutions and possibly a range of supported resolutions that we can also specify.

So, for example, if PrGeneral told us that we were capable of printing our 300-dpi image at 300 dpi, we would set the printer port's resolution to 300 dpi x 300 dpi by using PrGeneral with the setRsl opcode. Then all we'd need to do would be to draw the image at its original size. That's the easy case.

If we're printing to a device none of whose supported resolutions match our image's resolution, the best choice is usually the pair of horizontal and vertical resolutions that when multiplied yield the largest product. We'll need to scale the image to that resolution before printing. While this method of choosing resolutions isn't foolproof, it should typically give us the best results. Of course, if someone comes out with a driver for a printer that supports a resolution pair such as 600 dpi x 72 dpi, where there's a big difference between the horizontal and the vertical resolution, there might be problems with such an approach. Many times, we'll want the horizontal and vertical resolutions to be equal. The section on setting resolution under "Printing in Black and White" later in this article discusses this further.

We'll probably also want to put a ceiling on the resolution of the printer port. Otherwise, if we're printing to a Linotronic we may have to scale our 72-dpi images up about 3528 percent to 2540 dpi, and that will take a long, long, long time to print and require an enormous amount of memory. Of course there may be times when 2540 dpi is exactly what we want. We can always provide the user with a list of supported output resolutions to choose from.

Finally, suppose that we can't set the printer resolution because we're using a driver that doesn't support PrGeneral. We can tell this because after our call to PrGeneral, ResError is set to resNotFoundErr. In this case, we have only one recourse — to scale the image to the port's default resolution, 72 dpi.

Putting all this together, we end up with the GetBestDPI routine in the Final Adventure sample for setting the best resolution with PrGeneral. GetBestDPI obtains the best horizontal and vertical resolutions to use for printing with the selected driver. The function looks like this:

```
void GetBestDPI(short *pxDPI, short *pyDPI, short xDPI_ceiling,
                        short yDPI_ceiling, Boolean wantSquareDPI);
```

The caller places an ideal resolution pair (what the caller really wants to use) in the parameters pxDPI and pyDPI. This is also where the routine returns the resolutions it decides on. In xDPI_ceiling and yDPI_ceiling, the caller places the maximum resolution desired in either direction. For example, if you didn't want values larger than 300 dpi returned, you'd put 300 in both of these parameters. If wantSquareDPI

The LaserWriter's physical resolution is 300 dpi but printer drivers on the Macintosh return a 72-dpi port by default, because 72 dpi is the native resolution of QuickDraw. It's important to realize that unless you explicitly set the port's resolution to 300 dpi, you're working in a 72-dpi port and the effective resolution is cut by more than three quarters.•

is true, only square resolutions (those with equal horizontal and vertical components) will be considered.

The printer driver is expected to be closed upon entry to this routine and is therefore opened and closed around the PrGeneral code. If PrGeneral isn't supported by this driver, or if an error occurs, the routine returns 72 x 72 dpi, which is the default for Macintosh printer drivers. If the ideal resolution the caller passes in is available, we choose that, ignoring wantSquareDPI, xDPI_ceiling, and yDPI_ceiling. We figure that the calling routine knows more about the ideal resolution it requests than we do. Here's the code:

```
void GetBestDPI(short *pxDPI, short *pyDPI, short xDPI_ceiling,
                  short yDPI_ceiling, Boolean wantSquareDPI)
{
   TGetRslBlk     getResRec;
   Boolean        exactMatch = false;
   short          bestResX, bestResY, xDPI, yDPI,
                  desiredResX, desiredResY, rec;

// Open the driver for our PrGeneral call. Assume we'll return 72 x 72
// dpi until we find otherwise, and also store the desired resolution
// that the caller passed to us through the pxDPI and pyDPI parameters.
   PrOpen();
   bestResX = bestResY = 72;
   desiredResX = *pxDPI;
   desiredResY = *pyDPI;

   if (!PrError())
   {

// Ask PrGeneral for the resolution records for this driver.
      getResRec.iOpCode = getRslDataOp;
      PrGeneral((Ptr) &getResRec);

      if ((!ResError()) && (!getResRec.iError))
      {

// First check for the exact resolution pair that the caller requested.
// To begin with, check the range of resolutions supported to see if the
// pair is within that.
         if ((getResRec.xRslRg.iMin <= desiredResX) &&
               (getResRec.xRslRg.iMax >= desiredResX) &&
               (getResRec.yRslRg.iMin <= desiredResY) &&
               (getResRec.yRslRg.iMax >= desiredResY))
            exactMatch = true;
```

```
// If we didn't find an exact match, check the driver's discrete
// resolutions to see if we have one there.
         for (rec = 0; (!exactMatch) && (rec < getResRec.iRslRecCnt);
               rec++)
            if ((getResRec.rgRslRec[rec].iXRsl == desiredResX) &&
                  (getResRec.rgRslRec[rec].iYRsl == desiredResY))
               exactMatch = true;

// If we found an exact match, use it. Otherwise, loop through each
// resolution record and find the one that best matches our
// criteria.
         if (exactMatch)
         {
            bestResX = desiredResX;
            bestResY = desiredResY;
         }
         else
            for (rec = 0; (rec < getResRec.iRslRecCnt); rec++)
            {
               xDPI = getResRec.rgRslRec[rec].iXRsl;
               yDPI = getResRec.rgRslRec[rec].iYRsl;

               if ((xDPI <= xDPI_ceiling) && (yDPI <= yDPI_ceiling) &&
                     (!wantSquareDPI || (xDPI == yDPI)) &&
                     ((xDPI * yDPI) > (bestResX * bestResY)))
               {
                  bestResX = xDPI;
                  bestResY = yDPI;
               }
            }
      }
   }

// Return the best resolution pair we found and close the driver.
   *pxDPI = bestResX;
   *pyDPI = bestResY;
   PrClose();
}
```

The following code returns a rectangle to use when scaling from an image's bounds (srcRect) and resolution (ixDPI, iyDPI) to a printer port's resolution (pxDPI, pyDPI). The resulting rectangle (scaleRect) will have a top left corner of (0, 0).

```
void GetScaleRect(Rect *srcRect, short ixDPI, short iyDPI, short pxDPI,
                  short pyDPI, Rect *scaleRect)
```

```
{
   Fixed    scale;

   *scaleRect = *srcRect;
   OffsetRect(scaleRect, -scaleRect->left, -scaleRect->top);

   scale = FixRatio(pxDPI, ixDPI);
   scaleRect->right = FixMul(scale, (long) scaleRect->right <<16) >>16;
   scale = FixRatio(pyDPI, iyDPI);
   scaleRect->bottom = FixMul(scale, (long) scaleRect->bottom <<16) >>16;
}
```

BANDING THE IMAGE THROUGH A GWORLD

Pictures can include information that a printer driver can't understand, such as transfer modes and structures that have been added to the system since the driver was developed, and sometimes a driver can't reproduce certain operations that work great on the screen. For example, PostScript doesn't understand the concept of transfer modes, so the LaserWriter driver doesn't know what to do when it encounters such modes as blend, ditherCopy, and addMin. Aside from transfer modes, certain QuickDraw operations aren't supported by all drivers. For instance, CopyMask doesn't work with any of Apple's printer drivers as of this writing.

The upshot is that if you only use DrawPicture, some pictures are bound to print incorrectly on various printers because of operator incompatibility. The PICT named Incompatibility Test in the sample code folder demonstrates this problem. Try printing the picture with TeachText and comparing the output to the screen image. A safer approach to printing an image (although one that may require more data to be sent to the printer and thus result in slower printing) is to always send 32-bit-deep data to the printer by banding the image through a GWorld. Of course, if you know your application never needs 32-bit pixMaps, you can just use a GWorld deep enough for the data you'll be printing.

Here's how it works: Create a 32-bit-deep GWorld that has room for one horizontal (or vertical) strip of data of some arbitrary size. In the following example, we use horizontal strips. Call SetGWorld on this GWorld and then DrawPicture, passing the full image's picFrame. All of the picture outside the banding GWorld's bounds rectangle is clipped. The code might look like this:

```
#define BAND_HEIGHT 144    // 2 inches at 72 dpi.

pictRect = (*imgPICT)->picFrame;
bandRect = pictRect;
bandRect.top = 0;
bandRect.bottom = BAND_HEIGHT;
```

```
err = NewGWorld(&bandGWorld, 32, &bandRect, nil, nil, 0);

if (err == noErr)
{
   SetGWorld(bandGWorld, nil);
   destPix = GetGWorldPixMap(bandGWorld);

   LockPixels(destPix);
   DrawPicture(imgPICT, &pictRect);
   UnlockPixels(destPix);
}
```

This results in a band of the original picture being drawn to bandGWorld, which in turn can be copied to the printer port, like so:

```
SetPort(&(printPort->gPort));
srcPix = GetGWorldPixMap(bandGWorld);

LockPixels(srcPix);
CopyBits((BitMap *) *srcPix, &(printPort->gPort.portBits), &bandRect,
         &bandRect, srcCopy, nil);
UnlockPixels(srcPix);
```

To create the next band, shift bandGWorld's bounds rectangle down by one bandwidth and repeat the process. For best results, you may want to increase the printer port's resolution with PrGeneral, draw into a GWorld of the same resolution, and then use CopyBits to draw that in the printer port.

When you're working with 32-bit images, it's very useful to implement some sort of banding or picture spooling algorithm, since 32-bit images take up an enormous amount of memory, especially when you need to scale them to higher printer resolutions. All of the program samples on the CD have routines that implement banding and spooling. These routines also handle the special problems introduced when you need to dither and scale during banding.

When you send 32-bit-deep data to the printer driver, you inadvertently solve another problem as well — worrying about the printer's output characteristics. Printing images as 32-bit deep will give you the best output on all color printers whose drivers return a cGrafPort. You can be sure that when you send 32-bit-deep data the driver and printer will do the right thing — either print the image 32 bits deep or map it to the device's characteristics, be it an 8-bit device or whatever. You don't need to worry about checking the depth of the printer port or getting its GDevice or color table, which would be futile anyway since the port probably isn't a true cGrafPort.

OF COLOR TABLES AND THE LASERWRITER DRIVER

A word to the wise: The LaserWriter driver changes your image's color table. You must be prepared for this and know how to prevent its altering your printout.

Suppose you have an 8-bit color image with a custom color table. What happens when you print this with the LaserWriter driver using CopyBits when Color/Grayscale is selected? The driver returns a cGrafPort at PrOpenDoc time. As the drawing begins, the driver makes a copy of your image's color table. It then replaces the first entry in the color table with the current background color, and the last entry with the current foreground color. Once the foreground and background colors have been placed in the color table, the driver sends the image to the printer, passing the indexed RGB value for each pixel.

This means that if your foreground color is not the same as your last color table entry, or your background color is not the same as your first entry, your image may be altered when it prints. The best way to avoid this problem is to keep white in the first color table entry and black in the last, and make sure to set the foreground color to black and the background color to white before drawing.

Because the driver alters your color table, it's not a good idea to invert an image by inverting its color table, as some applications do. Imagine that you have an 8-bit grayscale image of a scanned photograph. Let's say that you want to print an inverted copy of the image and that its color table is a linear ramp of grays, from white to black. The easy — but incorrect — approach is to invert the entries in the image's color table and then print the image. The correct approach is to use CopyBits to copy the image over itself using notSrcCopy mode before printing.

Figure 1 compares printouts of an image inverted correctly and incorrectly. Notice that the incorrect method hasn't inverted absolute (or pure) black or white pixels in the image.

Why does the driver alter your color table? Because it's attempting to perform bitmap colorization. This is a feature of CopyBits that's not very well documented and that the LaserWriter driver supports. The version of CopyBits in System 7 will actually colorize an entire pixMap, although the LaserWriter driver has never been upgraded to support this functionality. The improvements to CopyBits colorizing are discussed in "QuickDraw's CopyBits Procedure: Better Than Ever in System 7.0" in *develop* Issue 6 and in Chapter 17 of *Inside Macintosh* Volume VI.

Original

Inverted correctly

Inverted incorrectly

Figure 1
Grayscale Image Inverted Correctly and Incorrectly

In general, if you don't know whether an image is 32 bits deep or 8 bits deep, you should print it at 32 bits. This way, you won't lose any color information. Of course, printing 32-bit-deep images means increased printer data and print times, so you may want to let the user have some control over the decision. Getting the best output may not be as important to a user as seeing an 8-bit draft of the image sooner.

PRINTING IN COLOR ON THE IMAGEWRITER

Most printer drivers today have been updated to return cGrafPorts when color ink is used. The only exception to this rule that I know of is the ImageWriter driver. Because all Apple ImageWriter drivers through version 7.0 return a grafPort, we can't rely on Color QuickDraw calls and structures to give us accurate color images when we have a color ribbon installed. We can draw only eight colors into a grafPort (traditionally called "the original QuickDraw colors").

Printing on the ImageWriter with DrawPicture works perfectly well as long as our picture is made up of original QuickDraw objects (those that appear in *Inside Macintosh* Volume I), each preceded by a call to ForeColor to set the foreground color to one of the eight original QuickDraw colors. For example, the following code will print correctly on an ImageWriter with a color ribbon, whether it's simply sent to the printer port or enclosed in a PicHandle that's then printed with DrawPicture:

```
SetRect(&bounds, 20, 20, 120, 120); // Initial object bounds (a square).
BackColor(whiteColor);              // Set background color to white.
ForeColor(cyanColor);               // Set foreground color to cyan.
FillRect(&bounds, gray);            // Fill square with 50% cyan pattern.
OffsetRect(&bounds, 70, 70);        // Move down a bit.
ForeColor(blackColor);              // Select black.
FrameRect(&bounds);                 // Draw a black square frame.
ForeColor(cyanColor);               // Select cyan.
PaintOval(&bounds);                 // Draw a cyan circle in the frame.
```

The result is shown in Figure 2. Without the calls to ForeColor, our picture would be recorded using our current foreground color for all objects. This is usually black and would cause everything to print as black.

If we need to print Color QuickDraw objects on an ImageWriter with a color ribbon, we must first convert them to original QuickDraw objects. In the case of pixMaps, we convert all of the pixMap's colors to the eight original QuickDraw colors and make a bitmap separation of the image for each color. The Color ImageWriter Adventures sample demonstrates how to do this.

CONVERTING TO THE ORIGINAL QUICKDRAW COLORS

First, possibly through banding, we use CopyBits to ditherCopy the source picture into a 4-bit GWorld whose color table is made up of the eight original colors. We

The original QuickDraw colors and their predefined constants are listed on page 158 of *Inside Macintosh* Volume I.•

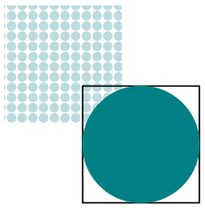


Figure 2
Product of Drawing With a Sequence of Calls to ForeColor

obtain this color table by passing a value of 127 to GetCTable, as explained in *Inside Macintosh* Volume V, page 81.

If we don't use ditherCopy, the resulting output will have colors determined by threshold comparison. In other words, every color in the original will simply be mapped to one of the eight original QuickDraw colors. This method will make scanned images look fake or "painted," which is not what we're looking for. In most cases, we'd rather have a dithered image that approximates more than eight colors by putting different colors side by side. Since we're printing only eight real colors, dithering is a necessity when using this method. For the curious, the Color ImageWriter Adventures sample allows you to turn dithering off for comparison.

MAKING THE SEPARATIONS

Once our image has been copied to the 4-bit "original color" GWorld, we can start making our separations. We need a Color QuickDraw searchProc that returns the position indicator for black or white, depending on whether or not the color passed matches the color we're looking for. If it does, the routine returns black. Since we'll be copying to a bitmap (in which a 0 pixel value indicates the background color and a 1 pixel value indicates the foreground color), this is all the code it takes:

```
pascal Boolean OQDSearch(RGBColor *anRGB, long *position)
{
   *position = 0;       // Initially assume no color.

   if ((anRGB->red == (*gOrgQDCTab)->ctTable[gCurColor].rgb.red) &&
         (anRGB->green == (*gOrgQDCTab)->ctTable[gCurColor].rgb.green) &&
         (anRGB->blue == (*gOrgQDCTab)->ctTable[gCurColor].rgb.blue))
      *position = 1;    // Color it.

   return true;         // To indicate that we've handled the color
                        // processing.
}
```


Color QuickDraw searchProcs are discussed in *Inside Macintosh* Volume V, pages 145–147.•

We'll make seven separations (one for each of the eight original QuickDraw colors except white). The code that follows is adapted from the Color ImageWriter Adventures sample and stores the different separations in a picture that uses only original QuickDraw primitives, so it can be sent with DrawPicture to the ImageWriter driver's grafPort with great results.

The process goes like this: Once we have the dithered image in our 4-bit GWorld, we create a 1-bit GWorld using exactly the same dimensions. We'll use this 1-bit GWorld to create our bitmap representations of each color separation. After setting the current GWorld to our 1-bit GWorld, colorSep, we call OpenPicture. This is critical because OpenPicture and OpenCPicture tie each open picture to the current port. (That's why you can have multiple pictures open at once as long as they're in different ports.) If we change ports, we can draw all we want and the calls will not be recorded into our picture. Only when we make the colorSep GWorld the current one will this picture's recording be enabled. Very cool.

```
PicHandle SeparateColors(PicHandle wPICT, Fixed scaleAmt,
                         Boolean useDither)
{
   QDErr          err;
   GWorldPtr      savedGW;
   GDHandle       savedGDH;
   PicHandle      sepsPICT = nil;
   Rect           pictFrame;
   GWorldPtr      OQDGWorld = nil;
   PixMapHandle   srcPix, destPix;
   GWorldPtr      colorSep = nil;
   short          QDColor[7] = {blackColor, yellowColor, magentaColor,
                                redColor, cyanColor, greenColor,
                                blueColor};

// Save the current GWorld and GDevice.
   GetGWorld(&savedGW, &savedGDH);

// Set our global color table to the eight original QuickDraw colors and
// get the picture's frame.
      gOrgQDCTab = GetCTable(127);
      pictFrame = (*wPICT)->picFrame;

// Create a 4-bit GWorld that uses the eight original QuickDraw colors.
// If there are no errors, band the picture, using ditherCopy if desired
// and scaling the amount we need to. The result is a representation of
// the image in the eight original colors.
      err = NewGWorld(&OQDGWorld, 4, &pictFrame, gOrgQDCTab, nil, 0);
      if (!err) err = BandPicture(wPICT, OQDGWorld, scaleAmt, useDither);
```

```
// Create a new 1-bit GWorld for the separations.
      if (!err)
      {
         err = NewGWorld(&colorSep, 1, &pictFrame, nil, nil, 0);

// Set the current GWorld to the 1-bit GWorld and create a picture.
// Note that this means that the picture is tied to the 1-bit GWorld.
// Only when that GWorld is current will data be recorded into the
// picture.
         if (!err)
         {
            SetGWorld(colorSep, nil);
            srcPix = GetGWorldPixMap(OQDGWorld);
            LockPixels(srcPix);
            destPix = GetGWorldPixMap(colorSep);
            LockPixels(destPix);

            ClipRect(&pictFrame);
            sepsPICT = OpenPicture(&pictFrame);
```

With the picture opened, the separations can be made. We go through each of the eight original colors (except white) and create a separation for that color. To do this, we set the current GWorld to one that's different from our picture's GWorld (to turn off recording). Next we install our searchProc and we CopyBits from the 4-bit GWorld to the 1-bit one. This gives us black bits only where the color in the original matches the current separation color. Then we delete the searchProc and set our current GWorld back to the 1-bit one. This reenables recording into our picture, and we record the foreground color for the current separation, followed by the separation's bitmap using srcOr mode. After all seven passes have been completed, we will get a picture with seven separations in it, overlaying each other to make the composite, which will differ slightly from the original because we lose some information to dithering. (See Figure 3.) We use srcOr mode so that the white is transparent; otherwise the white for each layer would overwrite the color from the previous layer.

```
            for (gCurColor = 0; (gCurColor < 7) && !err; gCurColor++)
            {
               SetGWorld(savedGW, savedGDH);
               AddSearch(OQDSearch);
               CopyBits((BitMap *) *srcPix, (BitMap *) *destPix,
                        &pictFrame, &pictFrame, srcCopy, nil);
               DelSearch(OQDSearch);

               SetGWorld(colorSep, nil);
               ForeColor(QDColor[gCurColor]);
```

```
                CopyBits((BitMap *) *destPix, (BitMap *) *destPix,
                        &pictFrame, &pictFrame, srcOr, nil);
            }

// Close the picture, restore our saved GWorld/GDevice, and dispose of our
// GWorlds and the global color table. Finally, return the picture we
// created.

            ClosePicture();
            SetGWorld(savedGW, savedGDH);
            UnlockPixels(srcPix);
            UnlockPixels(destPix);
         }
      }

   if (gOrgQDCTab) DisposeCTable(gOrgQDCTab);
   if (OQDGWorld) DisposeGWorld(OQDGWorld);
   if (colorSep) DisposeGWorld(colorSep);
   return sepsPICT;
}
```

After the seventh separation is made, we can jump into our print loop and print the image with DrawPicture. The result is a nicely dithered color image. For best results, we'd use PrGeneral to set the printer grafPort's resolution higher than 72 dpi. In the case of a printer such as the ImageWriter, though, this has a tremendous performance hit. Here's another good opportunity to provide some user interaction and let the user decide what to do, via a preferences setting or by adding an item to the print job dialog.

GOTCHAS

This method of printing a picture to an ImageWriter with a color ribbon will achieve great results without doing anything special. However, there are two gotchas with it.

First, if you generate an image at a high resolution and export it to another application, the printing application needs to know to call PrGeneral to boost the printer port's resolution. However, you can export the pictures at 72 dpi, use a picFrame that's correct for 72-dpi display of the image, or use OpenCPicture to store the resolution in the picture. In any of those cases, DrawPicture will do the right thing with the picture, even though the application doesn't. To see this, print out the sample PICT called Separations Test to a color ImageWriter using TeachText. TeachText has no special code to handle ImageWriter printing and yet it prints the PICTs generated by this method just fine. Pictures you create this way will print to a color ImageWriter from any application and can be pasted into word processors and such for color image output. Pretty neat, huh?

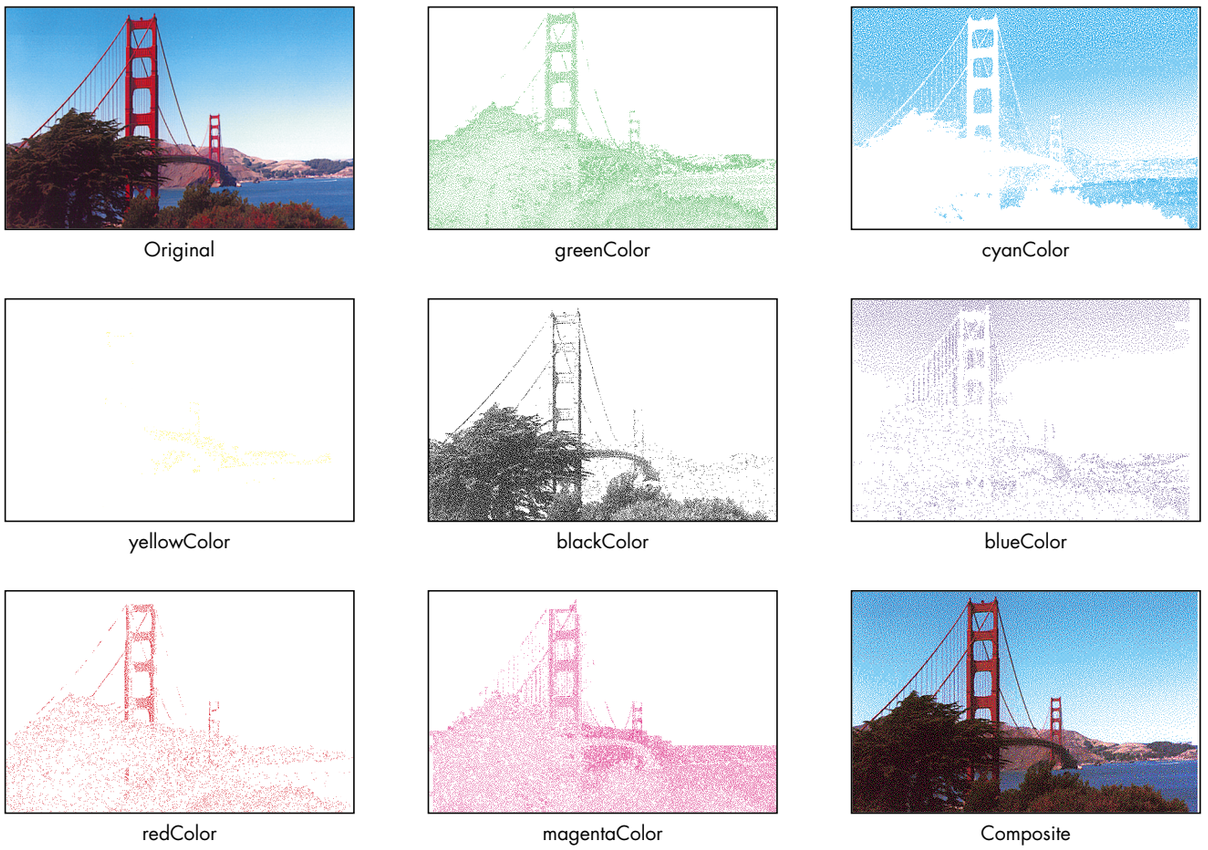


Figure 3
Seven Separations Created With Original QuickDraw for a Color Image

But, unfortunately, srcOr mode doesn't necessarily print well with all printer drivers. This means that these way-cool images may not print way-cool on printers other than ImageWriters. This isn't a problem in the sample code because we use this method only if we're printing on an ImageWriter. PICTs that are pasted into a document might be printed on any printer, however, so exporting these pictures could create more problems than it solves.

For more details on how to do the separations, see the Color ImageWriter Adventures sample. The sample prints color pixMaps using this method and allows you to specify high-resolution or low-resolution output. I strongly urge you to print at least one of the sample images using the application in ditherCopy mode and specifying high-resolution output. The results may surprise you, as they did me.

PRINTING IN BLACK AND WHITE

When we're printing pixMaps to a grafPort and the printer doesn't have (or the user doesn't want to use) color capability, we need to use dithering (or more precisely, a special kind of dithering called halftoning) to get any kind of decent output. In other words, we need to convert pixMaps to dithered bitmaps. The Halftone Adventures sample demonstrates three different dithering methods: the CopyBits ditherCopy method, the "true" halftone method, and the lazy person's halftone method. Before we look at these, a note about resolution.

SETTING RESOLUTION

When printing halftoned images, it's best to set the printer to a square resolution (equal horizontal and vertical dpi). The reason is that when we use mixed resolutions, our halftone matrix becomes distorted, and that can distort the printed image. This happens because dots that should be a fixed distance apart are now closer to each other in one direction than in the other.

We can compensate for this distortion when we create our halftone matrix, but it's likely to be a great deal of work, which is only marginally justified. All of the halftone routines in the sample code print using square resolutions. (They call the GetBestDPI routine described earlier with the wantSquareDPI parameter set to true.)

THE DITHERCOPY METHOD

The ditherCopy method uses CopyBits to dither the image to a 1-bit GWorld at device resolution and print that. If you're working with a device that has a low resolution (prints big dots) and a relatively constant physical dot size — such as the ImageWriter — then this method works fine. If, however, you're printing to a device that has a high resolution and a variable pixel size (from device to device, or even within the same device across time or due to variations in amount and type of toner, humidity, and paper type), this method results in image distortion. Figure 4 was dithered to a LaserWriter using this method, and the resulting distortion is very noticeable.

The distortion you see in Figure 4 is due to pixel error (the difference between the physical pixel drawn on the page and its size as the driver or rendering system models it). Since dithering must occur at device resolution, it's hard to compensate for the device pixel error when a dithered image is printed. Halftoning, on the other hand, increases the size of each dot, negating the pixel error that occurs during printing. Thus halftoning results in better output on devices such as the LaserWriter. This phenomenon is discussed further in "Making the Most of Color on 1-Bit Devices" in *develop* Issue 9 and is one of the main reasons that just doing a straight dither is not acceptable for most cases. The ditherCopy method does, however, provide a good benchmark to judge the other methods against.

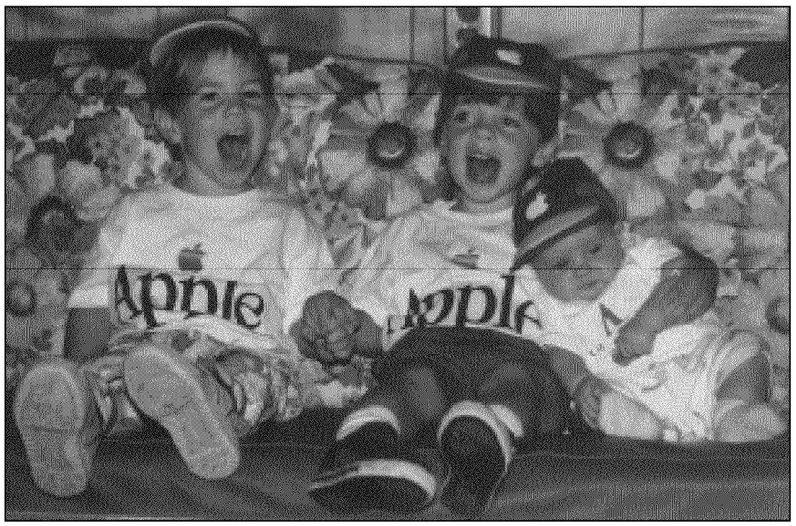


Figure 4
Distorted Sample Output From the ditherCopy Method

THE "TRUE" HALFTONE METHOD

The "true" halftone method is described in "Making the Most of Color on 1-Bit Devices" in *develop* Issue 9. You can read all about it there and try it out in the Halftone Adventures sample. Note that the routine in the sample code uses 8 x 8 halftones, but the algorithm described in the Issue 9 article is general and will work at any angle, any frequency, and any resolution. Also, since the sample's routine accepts only 8-bit-deep and 32-bit-deep pixMaps, the source image is passed in as a 32-bit-deep pixMap. When you use this sample code, an image may take one to two minutes to render before being printed, but the code can be optimized to increase its speed. Figure 5 provides an example of the kind of output we can expect using this method.

THE LAZY PERSON'S HALFTONE METHOD

I came up with the lazy person's halftone method to create fast "halftone-ish" output that looks very good and prints very fast. It works especially well on LaserWriters. Typical images render in 12 seconds or so (before printing), and I'm sure optimization would shorten this time. But note that this is not intended to be a general solution like the "true" halftone method; its usefulness is restricted to halftones at one angle, one frequency, and a square resolution.

Strictly speaking, this isn't halftone generation but rather halftone approximation with patterns. The difference is that in "true" halftoning, a halftone matrix is cookie-cuttered around the image, and adjacent pixels are taken into account when the

Figure 5
Sample Output From the "True" Halftone Method

halftones are created. In this way, the appearance of strong patterns (such as vertical stripes) can be removed. With the method I propose, the output appears to be a 0° 4 x 4 halftone, not a 45° 8 x 8 as in the Halftone Adventures implementation of the true halftone method. While this approach doesn't generate strong patterns, the absence of a 45° halftone is somewhat noticeable on lower-resolution printers like ImageWriters or those with drivers that don't support PrGeneral (and therefore must be used at 72 dpi).

Here's how it works: First, we dither the original image to a 4-bit grayscale GWorld, at 1/4 the optimal printer resolution. This may mean stretching or shrinking the original image. Next we find out how much of the printed image will fit on the paper. We use this information to limit the amount of data we're working with to just the pixels that will end up on paper. If the image extended 5 inches off the right edge of the paper, for example, it would be a waste of time to process that extra 5 inches. Once we have the dithered data and the bounds we're working with, we create a 1-bit GWorld that's four times as big as the 4-bit one. (This also means that it's at our printer port's resolution.) Going through the source (4-bit) image one pixel at a time, we create the halftoned output by matching up each pixel's index value with one of the patterns shown in Figure 6 and drawing that 4 x 4 pattern in our 1-bit GWorld.

For example, if we find a pixel has the index value of 8, the pattern with 8 dots in it is used. With 4 x 4 patterns, we could actually create 17 unique patterns (counting the pattern created when no dots are used). However, this wouldn't be helpful since our

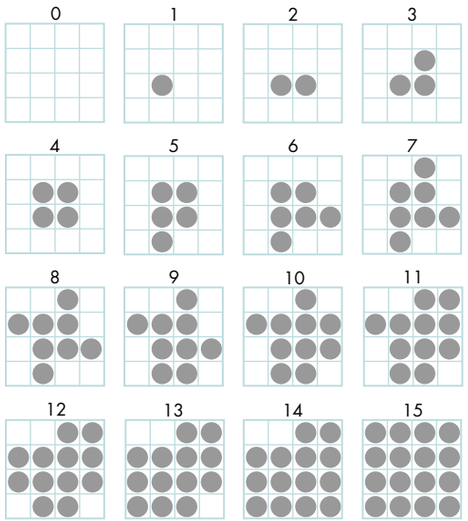


Figure 6
The Patterns Used to Approximate Halftones

image has only 16 shades of gray in it. Therefore, we ignore one, and I chose to drop the pattern for 15. (The pattern designated 15 is really one for 16.) The reason for using the pattern for 16 in the 15 spot is that black in our image will have a value of 15, and we want to make sure that black pixels are rendered as totally black patterns. Otherwise, the resulting image would have no solid black in it.

Once the entire image has been halftoned, we just CopyBits it to the printer port. Figure 7 provides an example of the kind of output we can expect using this method.

This method works especially well when we're printing at a high resolution. On the LaserWriter at 300 dpi, for example, the 4 x 4 patterns are so small (1/75") that they appear as a single dot. It's hard to believe that the output in Figure 7 was printed in just black! As you can see by comparing this output to the halftone output in Figure 5, there's very little difference between the two, and for speed considerations, the lazy person's method may be a viable alternative.

DETECTING THE PRINTING CONFIGURATION

We now have the methods that enable us to obtain high-quality output from the whole range of possible printing configurations when we print pixMaps. All we still need is a way to decide which method to use at print time.

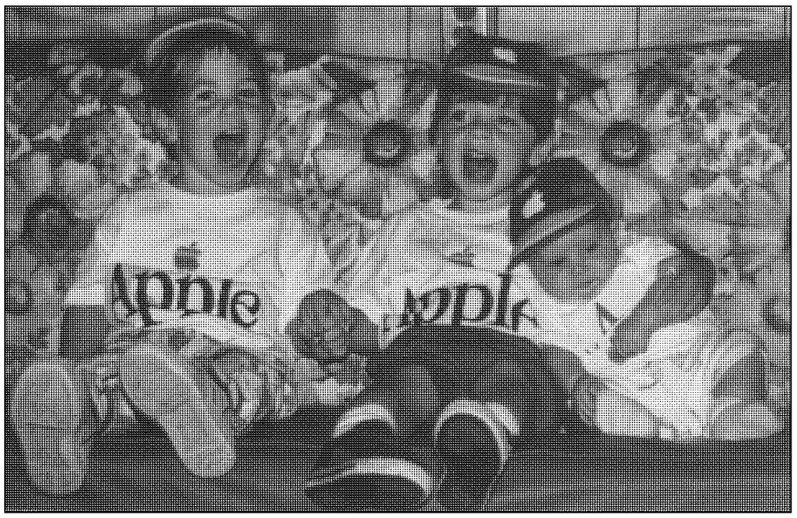


Figure 7
Sample Output From the Lazy Person's Halftone Method

To make this decision, we need to determine only three things:

- Do we have a cGrafPort or a grafPort?
- If a grafPort, is this an ImageWriter?
- If an ImageWriter, is a color ribbon installed?

That's it! These things can be determined in ways that will be compatible now and in the future. Let's take a quick look at the questions and how to determine their answers.

DO WE HAVE A CGRAFPORT OR A GRAFPORT?

We can determine whether we have a cGrafPort or a grafPort by checking the rowBytes value in the port returned by PrOpenDoc. If it's negative (the high bit is set), we have a cGrafPort. Otherwise, we have a grafPort. In C this reads:

```
printPort = PrOpenDoc(hPrint, nil, nil);
haveCGrafPort = (printPort->gPort.portBits.rowBytes < 0);
```

Since we'll probably need this information before we actually want to print (while we're still rendering), we may need to use the following routine. This routine assumes that we've already called PrOpen, that hPrint is a valid handle to a print

record, and that we're calling this routine outside of our PrOpenDoc/PrCloseDoc code. Normally, we would call this routine immediately after calling PrJobDialog.

```
Boolean HaveColorPrPort(THPrint hPrint, OSErr *anErr)
{
   Boolean     haveCGrafPort = false;
   TPPrPort    dummyPort;
   TPrStatus   statusRec;

   if (hPrint)
   {
      // Open a document and check for errors.
      dummyPort = PrOpenDoc(hPrint, nil, nil);
      *anErr = PrError();

      // If no errors, check the port's rowBytes value.
      if (*anErr == noErr)
      {
         haveCGrafPort = (dummyPort->gPort.portBits.rowBytes < 0);

      // We don't want to print yet, so kill the job by setting an error.
      // Clean up by closing the document and calling PrPicFile to delete
      // any spool file we may have created. Finally, clear the error
      // we set.
         PrSetError(iPrAbort);
         PrCloseDoc(dummyPort);
         if ((*hPrint)->prJob.bjDocLoop == bSpoolLoop)
            PrPicFile(hPrint, dummyPort, nil, nil, &statusRec);
         PrSetError(noErr);
      }
   }
   else
      *anErr = nilHandleErr;

   return haveCGrafPort;
}
```

The routine calls PrOpenDoc, checks the value of the returned port's rowBytes (negative means cGrafPort), and then posts an error to halt printing and calls PrCloseDoc. Finally, it calls PrPicFile to delete any spool file that may have been generated, clears the error it set, and returns true or false depending on whether or not the port we looked at was a cGrafPort. It's not glamorous, but it works.

If as a result of this inquiry we find that we have a cGrafPort, we give the go-ahead to printing with Color QuickDraw calls. If not, we go on to the next question.

IF A GRAFPORT, IS THIS AN IMAGEWRITER?

We can find out if we're talking to the ImageWriter driver by getting the high byte of a validated print record's prStl.wDev field. If the high byte is 1 or 5, we're using the ImageWriter or the ImageWriter LQ driver. In C:

```
#define IW_wdevID 1
#define IWLQ_wdevID 5
unsigned char  devID;

devID = (*hPrint)->prStl.wDev >>8;
if ((devID == IW_wdevID) || (devID == IWLQ_wdevID))
   /* Then we have an ImageWriter. */;
```

This method is described in the Macintosh Technical Note "Optimizing for the LaserWriter — Techniques" and is strongly discouraged there. So why am I suggesting that you use it? Well, unfortunately, there's no other reliable way to do this. In fact, checking the wDev has begrudgingly become an acceptable thing; developers have become so used to this method that we'd need to give ample warning before breaking it. However, you should expect that one of these days, checking wDevs will not be supported anymore. As soon as Apple provides a better method, you should jump on the code conversion bandwagon and replace all your wDev-snooping code.

It's important to make this check *after* the cGrafPort check because there are third-party printer drivers for the ImageWriter that support 8-bit color through cGrafPorts. If we first check for an ImageWriter and then jump to the ImageWriter grafPort printing code, we may be sacrificing output quality, since we may have been able to print using the Color QuickDraw methods described for cGrafPorts.

Anyway, if we find that we have an ImageWriter, we go on to the next question. Otherwise, we assume we have a monochrome printer and we accordingly launch the halftoning routine for printing.

IF AN IMAGEWRITER, IS A COLOR RIBBON INSTALLED?

In the case of the ImageWriter, we have two options for determining whether a color ribbon is installed: we can either ask the printer or ask the user.

To ask the printer, we would go through the serial driver if the ImageWriter were connected to a serial port or through AppleTalk Printer Access Protocol (PAP) if the printer were an AppleTalk ImageWriter. But this approach has a few problems. First, even if the user has a color ribbon, he may not want to use it. He may be printing rough copies of his work and want to save the color ink until he's ready to make a final copy. Or he may know that his color ribbon is worn out and prints well only in black. A second problem is that the printer must be turned on and selected when we

query it, or we'll hang until we time out. The delay is likely to thoroughly annoy our users.

Third, there's a problem with ImageWriter I support: the "ESC ?" query sequence (see the *ImageWriter Technical Reference Manual*) that's used to ask a serial ImageWriter if it has a color ribbon is not supported by the ImageWriter I. This means our query routine will hang until it times out, and we still won't know whether the printer has a color ribbon. A final and more compelling argument against performing the color ribbon query is that the methods that work today are unlikely to work under QuickDraw GX. Whether or not you decide to take advantage of QuickDraw GX's abilities, you should avoid implementing code that will make your application incompatible with it.

So we're left with the option of asking the user. The easiest way to do this is through a preferences setting. A slightly more coding-intensive but preferred approach is to add controls to the print job dialog. This might be a checkbox that simply says "Print in color," a pop-up menu that offers color or black and white (as in the Apple IIGS ImageWriter driver version 4.0), or, as I chose in the Final Adventure sample code, radio buttons for color or halftone output.

Even with this method, there are a few problems. If we add the control to every printer driver's job dialog, it will appear even when printers return cGrafPorts, in which case we'll want to ignore the setting. Also, if a checkbox is added to a driver like the 7.0 LaserWriter driver, the user will see redundant settings: a set of radio buttons for Color/Grayscale versus Black & White printing, and another checkbox for "Print in color." The way to get around this problem is to add the output controls only when the ImageWriter or ImageWriter LQ driver is being used, something we've already discussed how to determine. If we implement this solution, we'll want to store the last selected value for the control and default to it whenever the dialog is displayed. That will spare users from possibly having to click an extra button every time they print. However, if they change ImageWriters between print jobs, the saved flag may be incorrect for the new printer. This is a minor glitch that will become apparent the next time they print.

The bottom line here is that if we determine that our application is dealing with an ImageWriter with a color ribbon installed, we print using the eight original colors. Otherwise, we use our halftoning routine and print in black.

PUTTING IT ALL TOGETHER

To see how this decision process translates into code, take a look at the DoPrint routine in the Final Adventure sample on the CD. That sample rolls together into one neat package all the methods we've discussed in this article. Study it and give it a try to see how it works.

Adding controls to the print job dialog is described in the Macintosh Technical Note "How to Add Items to the Print Dialogs" and illustrated by PDlog Expand in the Snippets folder on the *Developer CD Series* disc.•

LAST WORDS

In this article, we've looked at the problems associated with color printing under the current printing architecture. We've seen that there's a real need for application developers to provide color printing support in their applications. We've also looked at techniques for printing high-quality representations of pictures containing pixMaps. These techniques consist of banding images through GWorlds for color-capable printers and drivers, creating color separations for printing on ImageWriters with color ribbons, and creating dithered halftones for black-and-white output.

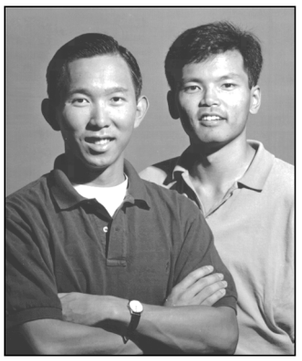
I mentioned that these techniques aren't intended for printing pictures that contain text, because when text is converted to pixMaps, all of the font information is lost, and the result is chunky, poor-quality text that's hard to read. You should always draw text separately from bitmaps or pixMaps, if at all possible. One way to do this is to write a routine to split a picture into two pictures: one with pixMaps, bitmaps, and foreground colors, and the other with everything except pixMaps and bitmaps (we'd want foreground colors in both). Once you have the two pictures, you can render the first using the methods discussed in this article and the second with DrawPicture. The order is important if we want the text to appear on top of the pixMap data. Remember to scale both pictures to the grafPort's or cGrafPort's resolution during printing.

As more technologies make use of color on the Macintosh, and more scanners and jumbo color monitors are shipped, users are going to need a way to get realistic hard copies of their screen displays. And although the color capabilities of Apple drivers and printers will continue to improve in both the short and long term (through such technologies as QuickDraw GX, ColorSync, and new printer drivers), interim solutions such as the ones proposed here will be needed for some time to come.

RELATED READING

- "Making the Most of Color on 1-Bit Devices" by Konstantin Othmer and Daniel Lipton, *develop* Issue 9.
- "Print Hints From Luke & Zz: CopyMask, CopyDeepMask, and LaserWriter Driver 7.0" by Pete ("Luke") Alexander, *develop* Issue 8.
- "Print Hints From Luke & Zz: Color Printing With LaserWriter 6.0 Revisited" by Pete ("Luke") Alexander, *develop* Issue 6.
- "Meet PrGeneral, the Trap That Makes the Most of the Printing Manager" by Pete ("Luke") Alexander, *develop* Issue 3.
- Macintosh Technical Notes "Optimizing for the LaserWriter — Techniques" (formerly #72) and "How to Add Items to the Print Dialogs" (formerly #95).
- *Fundamentals of Interactive Computer Graphics* by J. D. Foley and A. Van Dam (Addison-Wesley, 1982). Pretty much the standard in computer graphics books.
- *Graphics Gems* edited by A. S. Glassner (Academic Press, 1990). *Graphics Gems II* edited by J. Arvo (Academic Press, 1991). Lots of quick routines to do neat image processing stuff without the brain-bashing.

THANKS TO OUR TECHNICAL REVIEWERS
Pete ("Luke") Alexander, Hugo Ayala, Dan Lipton, Konstantin Othmer, Sean Parent•



GRAPHICAL TRUFFLES

THE PALETTE MANAGER WAY

EDGAR LEE AND FORREST TANAKA

No part of the Macintosh graphics environment is more feared, hated, or misunderstood than the Palette Manager. The Developer Support Center gets many questions about it from people who don't have any idea how to get it to do what they want. We've seen many people just give up on the Palette Manager completely and instead use lower-level routines that are much more difficult to use but easier to understand quickly.

The Palette Manager is actually very simple. It has no complicated heuristics that only rocket scientists can understand. In this column, we'll show how the Palette Manager gets its job done, and we'll talk about a couple of issues that you'll have to deal with to make your palettes do what you want them to do. You'll see that the Palette Manager is both easy to understand and a very useful part of the Macintosh Toolbox.

Before you read this column, it would be a good idea to read the Palette Manager chapter (Chapter 20) of *Inside Macintosh* Volume VI, which lays down the terminology that we'll use here.

WHAT HAPPENS WHEN A PALETTE IS ACTIVATED

The critical job that the Palette Manager does is activate a palette. This happens whenever you call SetPalette or ActivatePalette for the frontmost window and whenever a window that has a palette is activated. When a palette is activated, the Palette Manager loads the palette colors into the screen's color table. How it goes about doing this is determined by the usage mode of each entry in the palette.

You indicate an entry's usage mode by setting a flag in its usage field. There are four usage modes: pmCourteous, pmTolerant, pmAnimated, and pmExplicit. You can choose a separate usage mode or combination of usage modes for each entry in a palette, or you can give all the entries the same usage mode. Let's take a look at what each usage mode is good for and what effect each one has when a palette is activated.

pmCourteous. The pmCourteous usage mode enables you to replace RGBColor records in your code with single integers. Thus, having a palette of courteous colors gives you an alternative way to specify foreground and background colors. This is great for localizers who might need to change the colors in your program to something more meaningful in other countries, and it's great for you if you feel like changing a color without recompiling.

Activating a palette of courteous colors simply tells the Palette Manager to use your window's palette as a sort of lookup table. When your window is the current port and you call PmForeColor or PmBackColor with a palette index, the Palette Manager simply retrieves the color in your window's palette at that index and uses it for any subsequent drawing to that window. Courteous colors never change the screen's color table — they get mapped to the closest colors already available there.

Here's an example: Without the Palette Manager, you would draw a green oval in a window by setting up an RGBColor record with a red component of 0, a green component of 65,535, and a blue component of 0 and passing this record to RGBForeColor; then you would call FrameOval. To change the oval to blue, you would modify your RGBColor record in your source code and recompile your program. Now, suppose instead you brought in the Palette Manager by setting up a palette resource (resource type 'pltt') that contained one courteous entry — green — with an index of 0. In your code, you would call PmForeColor(0) instead of RGBForeColor with green. When you called

EDGAR LEE (AppleLink EDGAR) Before Edgar's dog, Sunny, departed for the East Coast, we asked her if she could tell us a little about him. Here's what she had to say: "Edgar . . . is that his name? Oh yeah, nice human. A little hairless for my taste, but a good guy. He works over there in DTS or something. He used to come home late all the time. At first I thought he was seeing another dog, then I realized he's just a nerd. And how is he to me? Well, let's see, he takes care of me, entertains me. I bark once, he feeds me; I bark twice, we go out for a walk. Not bad for an owner; I've heard worse stories. Does he ever get upset with me? I suppose at times he does. I probably deserve it; carpet cleaning isn't cheap, you know. But hey, I see a clean spot, I go for it."•

FrameOval, the oval would be drawn in green. To change the color to blue, you would just use ResEdit to modify the green entry in your 'pltt' resource to blue.

So calling PmForeColor for a courteous palette entry is just like calling RGBForeColor, but instead of supplying the RGB components, you simply pass an index into the palette, where the index points to the color's RGB components (see Figure 1).

pmTolerant. The pmTolerant usage mode is used when you want to be sure that a specific set of colors is available to the screens that your window is on. It's a bummer to draw a rainbow in a window on a screen that another application has removed all the greens and yellows from. You need a palette of tolerant colors to assert your application's right to the colors it needs to display its images optimally. With such a palette, you can change the colors in a screen's color table to ones that you want.

Before we look at how a palette of tolerant colors is activated, realize that Color QuickDraw always wants white in the first entry of a screen's color table and black in the last. To enforce this rule, Color QuickDraw protects these two entries from being changed to other colors. That means a palette of tolerant colors can change all the colors of a screen's color table except two.

When a palette of tolerant colors is activated, the Palette Manager checks each entry in the palette and associates it with an entry in the screen's color table. Let's say we have a palette with three entries — bright green, black, and dark yellow — attached to a window on a 16-color screen. All three palette entries are tolerant, with a tolerance of 0. The Palette Manager does the following:

1. It checks the first entry in the palette, bright green, and searches the screen's color table for the same bright green. It finds that color near the middle of the color table, and so associates palette entry 0 with this existing bright green entry in the color table.
2. It searches the screen's color table for the second entry in the palette, black. It finds it at the very end, so palette entry 1 corresponds to entry 15 of the color table.
3. It searches the screen's color table for dark yellow. There isn't one, so it chooses a color table entry to change to dark yellow. It can't choose the black or the white entry because they're protected and can't be changed, and it can't change the bright green entry because that entry is already associated with entry 0 of the palette. So it chooses one of the other color table entries and changes it to dark yellow.

Because the color table has been changed, the Palette Manager makes sure that other windows are redrawn,

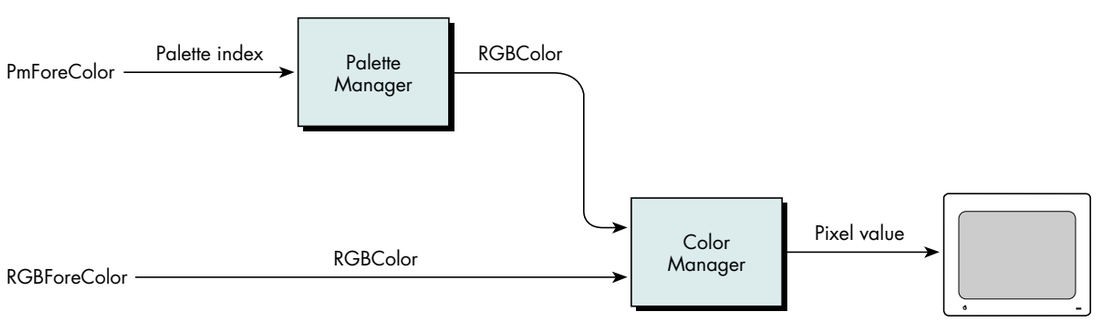


Figure 1
Alternative Ways to Specify a Foreground Color

FORREST TANAKA (AppleLink TANAKA) has spent the last couple of months learning how to be a domestic kind of guy. Once worried about paying the rent, he's now worried about paying the mortgage. Once worried about his downstairs neighbors, he's now worried about getting the best fertilizer. Now he's even the stepparent of an old dog and a cat with an attitude. As a final blow to his carefree days of youth, he has to mow the lawn!•

just in case they were drawn using the color table entry that was changed to dark yellow. It does this by sending update events to all windows as soon as the palette is activated. If no color table changes were needed, the Palette Manager doesn't bother doing this.

Once our three-entry palette has been activated, we can call PmForeColor, passing it 0, 1, or 2 to draw objects in bright green, black, or dark yellow, respectively. In fact, we could call RGBForeColor, passing it bright green, black, or dark yellow RGBColor records, and they would use the same colors that our palette loaded into the screen's color table. Figure 1 applies to palettes of tolerant colors as well as palettes of courteous colors.

If there are more palette entries than will fit in the screen's color table, the Palette Manager associates each palette entry with a color table entry until no more color table entries are available and then interprets the rest of the palette entries as courteous. For example, let's say a 20-entry palette is activated on a 16-color screen, where each palette entry is pmTolerant with a tolerance of 0 and neither black nor white is in the palette. Beginning with the first palette entry, the Palette Manager associates each entry with a color table entry. The 15th palette entry can't be associated with any color table entry because the black and white entries are protected from changes and all 14 other entries have already been associated with palette entries. So the 15th palette entry and all entries beyond it are simply treated as courteous colors.

pmAnimated. On indexed devices, the pmAnimated usage mode is used to do color table animation, which gives you smooth, fast visual effects simply by changing the colors in your screen's color table very quickly. You don't have to redraw anything to see this animation; you just use the Palette Manager to change the interpretation of the colors of your existing image. This is great for games and fast controls for image processing applications. On direct devices, animated entries are treated as courteous entries.

Like pmTolerant entries, each pmAnimated palette entry is associated with an entry in a screen's color table when the palette is activated, and the colors in the palette are put into the screen's color table. But changing color table entries for color animation changes everything on the screen that uses those same color table entries, like the desktop or window frames. That's usually not what we want, so the Palette Manager forces everything outside the window to be redrawn without the colors that are being used for color animation — those colors are off limits. In fact, the only way to use those colors is to call PmForeColor or PmBackColor for an animated palette entry and then draw some QuickDraw object. Remember, the big difference between tolerant and animated colors is that color table entries that are used for tolerant colors can be used by anyone, but animated color table entries are used only by objects drawn in the palette's window after a call to PmForeColor or PmBackColor.

Let's use our three-entry palette as an example again, but this time assume that each entry is animated. The Palette Manager first takes the bright green entry, picks a color table entry on the screen, and changes it to bright green. It doesn't matter if there's already a bright green in the color table. As usual, the Palette Manager avoids the black and white entries at either end of the color table. It then picks another color table entry and puts black into it, and does the same for the dark yellow entry. If you call PmForeColor(0) and draw an object, it's drawn in bright green. But if you call RGBForeColor for bright green and draw an object, it doesn't use the bright green that's been defined as animated. Instead, it uses the closest color to bright green available, aside from any color table entries that have been defined as animated.

pmExplicit. The pmExplicit usage mode is rarely used alone, and there's not much to it beyond what's described in the Palette Manager chapter of *Inside Macintosh* Volume VI. We'll discuss in the next section the more interesting case of using pmExplicit along with the other usage modes.

ARE BLACK AND WHITE NEEDED IN A PALETTE?

When attaching a palette to a window, the Palette Manager works in a way that affects whether you

Indexed and direct devices are discussed in the Graphics Overview chapter (Chapter 16) of *Inside Macintosh* Volume VI.•

should store black and white in the palette. We'll outline the way it works in two different categories. The first category applies to palettes containing the same number of entries as the screen's color table, and the second category applies to palettes containing fewer entries than the screen's color table.

Same number of entries in palette and color table. If the palette contains the same number of entries as the screen's color table, black and white should be stored in the palette. If these two entries aren't stored in the palette, the Palette Manager will ignore two entries in the palette when loading the palette colors into the screen's color table, to avoid overwriting the color table's black and white entries. The Palette Manager will decide which palette entries to ignore based on the usage field for each palette entry.

As an example, let's take the case of a palette all of whose entries are defined as pmTolerant + pmExplicit. Because the pmExplicit flag tells the Palette Manager to store each palette entry in its respective index in the screen's color table, the choice of which palette entry to ignore is fairly straightforward. The colors stored at the first and last entry of the palette correspond to the protected entries in the screen's color table, so these entries will be ignored.

In the case of a palette containing entries not defined with the pmExplicit flag set, the decision of which two palette colors to ignore can seem somewhat random. This is because the decision is based on the current distribution of the palette entries in the screen's color table, where the distribution is derived from the tolerance values of the palette entries and the existing colors in the screen's color table before the palette was activated.

For example, suppose we have a 16-entry palette and a 4-bit screen color table as shown in Figure 2 (we've used the default color table). Figure 3 shows how the screen's color table will look in two different cases when the palette of the frontmost window has been activated. In these figures, the explicit entries are

Figure 2
An Example Palette and Color Table

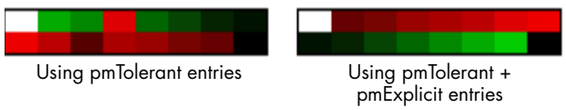


Figure 3
The Color Table With the Palette of the Frontmost Window Active

distributed sequentially in the screen's color table, whereas the nonexplicit entries are scattered throughout the color table.

For the explicit entries, we see that the first and last entries of the palette are not loaded into the screen's color table, to protect the color table's white and black entries. However, for the nonexplicit entries, the two palette colors ignored aren't necessarily the first and last entries of the palette. When determining where the nonexplicit palette entries should be stored in the color table, the Palette Manager first checks to see which colors in the screen's color table already match those in the palette. If there's a match within the specified tolerance, that palette entry is stored at the index of the matching color in the screen's color table.

And one other thing: When multiple nonexplicit palette entries match (within the specified tolerance) the same color in the screen's color table, all those palette entries are stored at the same index in the color table. This means that only one slot in the color table is needed rather than as many slots as there are palette entries.

Fewer entries in the palette. Now, if the window's palette contains fewer entries than the screen's color table, the palette entries' usage field plays a large part

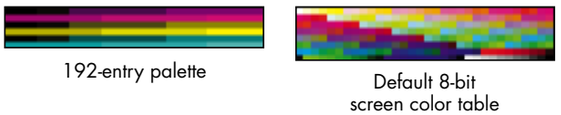


Figure 4
Another Example Palette and Color Table

Figure 5
The Color Table With the Palette of the Frontmost Window Active

in determining whether black and white should be included in the palette. The reason for this is similar to the previous case for nonexplicit entries.

If all the entries in a palette are defined without the pmExplicit flag set, the presence of black and white in the palette isn't as critical, since the palette entries will likely be scattered throughout the screen's color table while avoiding the protected white and black colors stored in the first and last slots of the screen's color table. Since there are fewer palette entries than color table entries, we needn't worry about palette entries getting ignored. So in this case, creating a palette without a black entry or a white entry is perfectly fine as long as there are enough slots in the screen's color table to hold all the palette entries and the two protected colors.

However, if the palette entries are all defined with the pmExplicit flag set, there's a good chance that one of the palette's entries will be ignored. And the palette entry that does get ignored will usually be the first entry in the palette, because this entry shares the same index as the protected white entry in the screen's color table.

For example, suppose we have a 192-entry palette and an 8-bit screen color table as shown in Figure 4 (again, we've used the default color table). Figure 5 shows how the screen's color table will look in two different cases when the palette of the frontmost window has been activated.

Again, the explicit entries are distributed sequentially in the screen's color table, starting with the first entry in the color table. Because the first entry in the screen's color table is protected from being overwritten, the first entry in the palette is ignored. But in the nonexplicit case, the entries are distributed somewhat differently. Depending on what colors are already in the screen's color table, the nonexplicit entries can be stored anywhere throughout the color table. And in this example, since there are clearly more slots in the screen's color table than needed by the palette entries, all the colors in the palette appear in the color table; none are ignored. So again in this case, including black and white in the palette really isn't necessary.

WHERE TO PUT BLACK AND WHITE IN THE PALETTE

We've seen how the way the Palette Manager works can affect whether you decide to store black and white in your palette. In all the instances we mentioned, the positions of the black entry and white entry were always the same: white first and black last. However, in certain cases, you may not want to position white first and black last.

In the case where you'd like to create just one palette to handle devices at multiple bit depths, the black and white entries should be stored as the first two colors in the palette. This ensures that the two colors used on a 1-bit device are present. Likewise, to ensure that the optimal colors are used at depths 2, 4, and 8, we do the same thing for each additional depth. We store the preferred colors at the appropriate position in the palette. Figure 6 shows how a typical palette could be configured to handle multiple bit depths.

In our sample palette, the first 16 colors are defined as shades of gray, because we've decided our window would look best when displayed in grayscale on a 1-, 2-, or 4-bit device. For the 1-bit and 2-bit devices, we

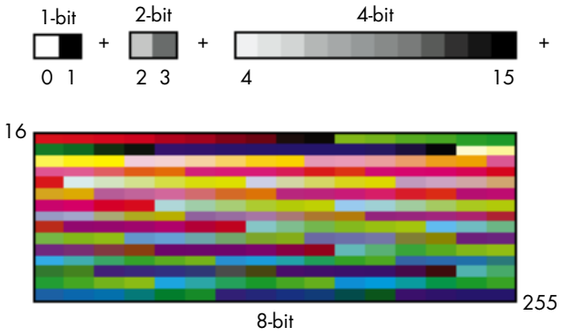


Figure 6
A Palette Configured to Handle Multiple Bit Depths

simply choose the appropriate shades for those depths and store them in the first four slots of our palette. But for an 8-bit device, we include as many colors as we can for the optimal display at that depth. For this example, we added the nongrayscale colors from the standard 8-bit color table to the remaining slots in our palette. Because the Palette Manager only uses the maximum number of colors it can (starting at the first index in the palette) for a specific bit depth, only the colors we want shown will be shown. Also, because the placement of the colors determines which colors are available at a certain depth, all the palette entries must be defined as explicit entries.

Another way of ensuring that certain palette entries are available at certain depths is to apply the inhibit usage categories to the palette entries. These inhibit constants tell the Palette Manager which entries are available under the current color environment. Depending on which inhibit constant is used, the palette entries can be inhibited from a specific bit depth and from a color or grayscale device. So by combining various inhibit constants to our sample palette, we can inhibit the colors outside the current depth's range from being used. In our example, if entry number 16 were defined with

```
pmInhibitC2 + pmInhibitC4 + pmInhibitG2 +
    pmInhibitG4
```

this entry would be available only on an 8-bit or deeper color or grayscale device.

ONE LAST WORD

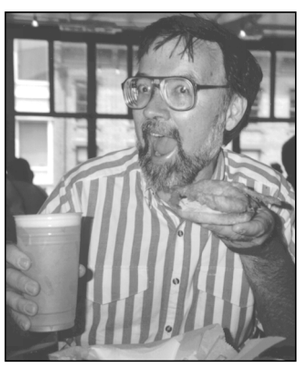
The Palette Manager works very simply, but it has so many options and effects that it can seem complicated. By understanding how the Palette Manager makes its decisions, you should find it easy to figure out how to make it do precisely what you want. We hope this column has made this clear, so that you can use the Palette Manager and avoid fussing with the alternatives.

The inhibit constants are discussed in the Palette Manager chapter (Chapter 20) of *Inside Macintosh* Volume VI.•

Thanks to Bill Guschwan, Shannon Holland, Guillermo Ortiz, Konstantin Othmer, Brigham Stevens, and John Wang for reviewing this column. Special thanks to Joseph Maurer and Faith Pai.•

DEVICELOOP MEETS THE INTERFACE DESIGNER

With the ascendancy of multimedia, 3-D shading and elaborate color backgrounds are showing up in an increasing number of interface designs. But what happens when these sophisticated interface elements must be displayed across multiple monitors of different bit depths? This article explains how to use the DeviceLoop function to take care of the device, clipping, and bit-depth logistics involved in multiple-monitor displays.



JOHN POWERS

One of the great things about the Macintosh is its ability to support more than one monitor at a time. You can display windows in any active monitor or split a window — and the objects in it — across several monitors at once. What's more, you can make an image adjust to the bit depth and other capabilities of each monitor it's displayed on, so that the visual interface looks as good as it possibly can on each of the devices attached to the computer.

I recently worked on a project in which one of the goals was exactly that — we wanted our application windows to look really good across multiple monitors and at any bit depth. The task was complicated by the fact that the interface was quite sophisticated graphically. To give our windows a distinctive, three-dimensional look, we used shaded color graphics. We filled the content area with background graphics, text, patterned and colored lines, and 3-D buttons. With the exception of our standard List Manager lists, all the window objects were drawn by our application program. Even the conventional scroll bar, close box, and zoom box were replaced by custom art drawn by the application, not the Window Manager.

Displaying these complex windows across multiple monitors was obviously going to be a challenge. We knew that the Finder, for example, pulled it off — whenever Finder windows span monitors of different bit depths, the parts of the window on each monitor are drawn to the individual monitor's depth. "If the Finder does it, so can we," I decided, although I actually knew very little about how to solve the problem.

JOHN POWERS (AppleLink JOHNPOWERS) started his career as a behavioral scientist, studying how people use computers. He worked his way up the management ladder, and then cofounded a company that developed software for the first home computers. That lead him to Atari, but Atari got weird, so John joined Convergent Technologies to develop the WorkSlate notebook computer, eight years before the PowerBook. That led him to another management ladder and into The Learning Company, where he developed software for children. Locked in his management office, John discovered the Macintosh and decided to become a Macintosh software developer. Now he's at Apple Computer developing Macintosh software that helps people use computers.•

DEVICELOOP TO THE RESCUE

I bit the bullet. The search for ways to draw a window across multiple monitors led in a number of directions, all of them involving visible regions, clipping regions, and region-rect conversions. I asked a lot of people for advice, and while everyone was gracious in offering help, the job was looking complicated. Fortunately, one of the advice givers suggested that I check out the DeviceLoop function in *Inside Macintosh* Volume VI. (I found out later that the advice giver was the author of the DeviceLoop function.)

When I looked up DeviceLoop in Volume VI, here's what I found:

> The DeviceLoop procedure searches all active screen devices, calling your drawing procedure whenever it encounters a screen that intersects your drawing region. You supply a handle to the region in which you wish to draw and a pointer to your drawing procedure. . . . If the DeviceLoop procedure encounters similar devices — having the same pixel depth, black-and-white/color setting, and matching color table seeds — it makes only one call to your drawing procedure, pointing to the first such device encountered.

This sounded exactly like what we were looking for. The Window Manager itself uses DeviceLoop to display window components on a variety of monitors. Since we were drawing our own windows, DeviceLoop was clearly what we needed.

Here's what DeviceLoop looks like in C:

```
pascal void DeviceLoop (RgnHandle drawingRgn,
                        DeviceLoopDrawingProcPtr drawingProc,
                        long userData, DeviceLoopFlags flags);
```

The drawingRgn parameter is a handle to the region that will be drawn in (usually a window's visRgn). The drawingProc parameter is a pointer to your drawing routine (see below). The userData parameter is a long that gets passed to your drawing routine. Finally, the flags parameter controls how devices are grouped before your drawing routine is called. (Pass 0 for the default behavior — grouping similar devices together. See the description in *Inside Macintosh* for other possible values.)

The drawing routine needs to be declared as follows:

```
pascal void MyDrawProc (short depth, short deviceFlags,
                        GDHandle targetDevice, long userData);
```

Here the depth parameter is the depth of the device you're currently drawing on. The deviceFlags parameter is a copy of the device's gdFlags, targetDevice is a handle to the device, and userData is whatever you passed to DeviceLoop.

The DeviceLoop call first appears in System 7. If your application will be running under an earlier version of system software, you'll need to implement your own DeviceLoop function. For an example of how to do this, see the column "Graphical Truffles: Multiple Screens Revealed" in Issue 10 of *develop*.•

DeviceLoop works like this: Each time your drawing routine is called, the current port's visRgn will have been set to the intersection of your drawing region and some screen device. DeviceLoop passes the drawing characteristics of the particular screen it's working on to the drawing routine, which can then make use of them — for instance, by drawing to the appropriate bit depth. In short, DeviceLoop takes care of all the device, clipping, and bit-depth logistics, while all you have to do is draw.

USING DEVICELOOP IN AN OBJECT-ORIENTED WORLD

In our application, we had to draw not only the contents of the window, but also the window itself. True to our object-oriented design, we created classes for all the interface objects. These classes included a TArt class for backgrounds, graphics, and 3-D button objects; a TLine class for lines; a TTxt class for black-and-white text; and a TBkg class for backgrounds for the text. Although we used DeviceLoop for drawing objects in every class except the text classes, the heart of the process is best illustrated by our use of DeviceLoop for TArt objects.

The graphics for TArt objects were stored as PICT resources. To give the best possible image, the interface designer created an 8-bit-deep PICT for display depths of 8 bits or deeper. For all other display depths and CPUs without Color QuickDraw, she created a 1-bit-deep, black-and-white PICT. We could have let the Macintosh use the 8-bit PICT for all drawing — color and black-and-white — and, with dithering, the results would have been pretty good. But since we had our own hand-designed, 1-bit version of the PICT, DeviceLoop was a better solution.

Our window object kept track of all the interface objects that it needed to draw. When an update event was received, the document object told the window object to draw. Specifically, our BeginUpdate/EndUpdate function called a particular drawing routine for each of the objects. Each object, in turn, called DeviceLoop with our DrawProc callback, which contained the actual drawing code for that object. Figure 1 shows this strategy.

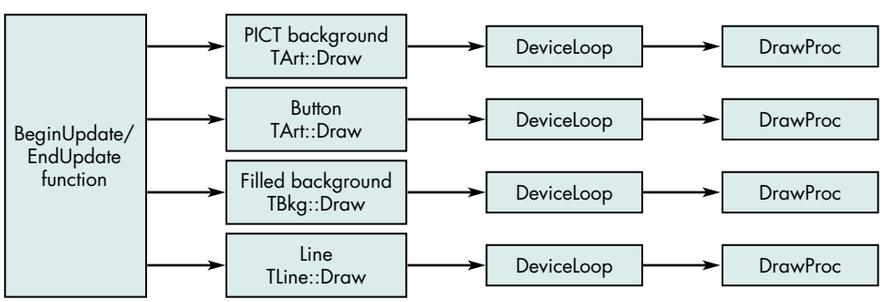


Figure 1
An Inefficient Way to Incorporate DeviceLoop

We used this DeviceLoop-within-each-object's-drawing-procedure approach until someone pointed out how inefficient it was to call DeviceLoop for every interface object. We realized that it would be much better to call DeviceLoop once and have the drawing procedure that we passed to it decide which object had to be drawn. We wound up with a single DeviceLoop call in the window's BeginUpdate/EndUpdate function, as shown in Figure 2. The use of a single DeviceLoop call in the window object really streamlined the design.

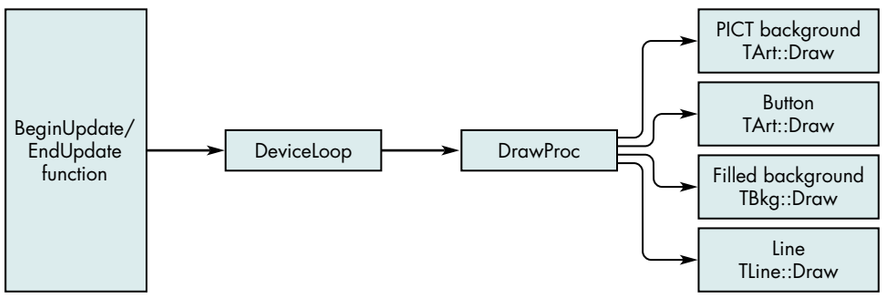


Figure 2
A Better Way to Call DeviceLoop

One problem we encountered was that the compiler balked whenever we referenced our drawing routine (called DrawProc) in the DeviceLoop parameter list. We even included the scope — TWin::DrawProc — and that didn't help. The breakthrough came when we made DrawProc static. Unfortunately, changing it to static caused another problem: the compiler choked when we referenced **this** within DrawProc. We forgot that static functions can't reference nonstatic member variables. (You C++ aficionados are probably smiling, but we recent converts must struggle at first.) We couldn't use static variables, however, because each of our objects required its own variables. Thus, to access an object's variables, we had to pass the window object pointer in the userData parameter of the DeviceLoop function.

AN EXAMPLE

The *Developer CD Series* disc contains a sample application that shows how we used DeviceLoop for TArt objects in our interface. The application, DeviceLoopInDrag, displays a window that can be dragged between monitors of different bit depths. Figure 3 shows this window spanning a grayscale and a black-and-white monitor.

Excerpts from the DeviceLoopInDrag source code follow. First there's the update function that's called whenever the window needs to be redrawn. It just calls the drawing procedure for the window object (TWin).

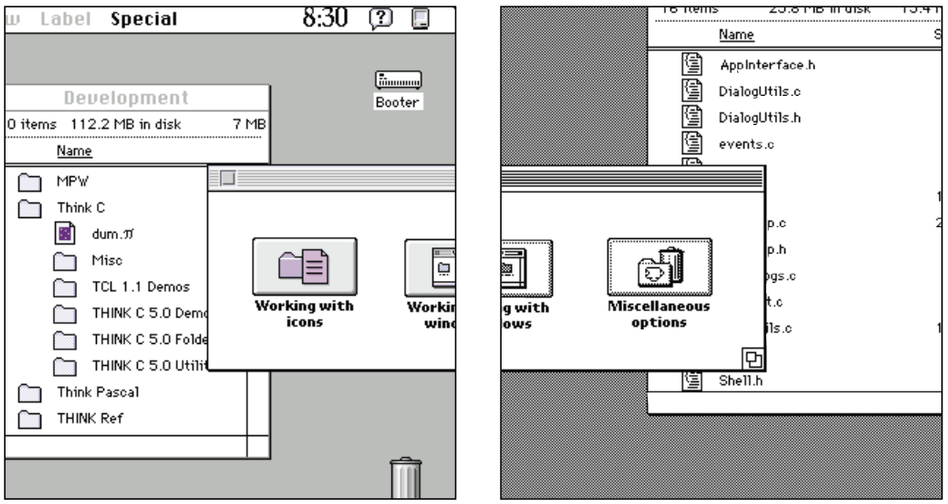


Figure 3
DeviceLoop in Action

```
// TDoc::DoUpdate
// Document object.
// Entry for update event action.
void
TDoc::DoUpdate()
{
   BeginUpdate(this->fDocWindow);
   this->fWinObj->Draw();
   EndUpdate(this->fDocWindow);
}
```

The window's drawing procedure does little more than set up and call DeviceLoop. Notice that we're passing the reference to the current window object — **this** — in DeviceLoop's userData parameter, as described earlier. Since we want the default DeviceLoop behavior, we set the flags to 0.

```
// TWin::Draw
// Window object.
// Within BeginUpdate/EndUpdate.
void
TWin::Draw()
{
   // Have DeviceLoop manage the drawing.
   // Pass the window object in userData.
```

```
   long              userData = (long)this;
   DeviceLoopFlags   flags = 0;
   GrafPtr           myPort;
   GetPort(&myPort);
   DeviceLoop(myPort->visRgn, TWin::DrawProc, userData, flags);
   // Draw the stuff we don't need DeviceLoop for.
   // We tell the subview to take care of that.
   this->fView->Draw();
};
```

Next, theTWin drawing procedure is the callback procedure that DeviceLoop invokes to coordinate the drawing of each of the elements on the screen.

```
// TWin::DrawProc
// Called by DeviceLoop.
// A static function. Must be in a resident segment, locked and
// unpurgeable. Because it's static, it can't access object member
// variables directly. We use the window object passed in userData
// to access its member variables.
#pragma segment Main
pascal void
TWin::DrawProc(short depth, short /*deviceFlags*/,
         GDHandle hTargetDevice, long userData)
{
   // Get the window object from userData.
   TWin* theWinObject = (TWin*) userData;
   // Use depth of 1 if we have a computer without Color QuickDraw.
   depth = (hTargetDevice==NULL)?1:depth;
   // Draw our objects.
   theWinObject->fBackground->Draw(depth);
   theWinObject->fLogo->Draw(depth);
   theWinObject->fText->Draw(depth);
   theWinObject->fButton->Draw(depth);
};
```

Finally, here's the actual TArt::Draw function, used for various items in the window. Based on the bit-depth parameter passed to it, the Draw function decides whether to use the black-and-white or the color version of its PICT.

```
// TArt::Draw
// All art objects (PICTs) are drawn here. This is where we distinguish
// between B&W or color renderings of TArt objects. The B&W rendering has
// a resource ID that's kBWOffset larger than its color counterpart value.
void
TArt::Draw(short depth)
```

```
{
   // Don't draw empty art.
   if(this->fPictID==0)
      return;
   PicHandle   hPict;
   if(depth<8)
   {
      // Use B&W PICT.
      hPict = (PicHandle) GetResource('PICT', this->fPictID+kBWOffset);
   }
   else
   {
      // Use color PICT.
      hPict = (PicHandle) GetResource('PICT', this->fPictID);
   }
   if(hPict)
   {
      Rect  theDrawRect;
      this->GetDrawRect(theDrawRect);
      HLock((Handle) hPict);
      DrawPicture(hPict, &theDrawRect);
      HUnlock((Handle) hPict);
   }
};
```

SUMMING UP

How did we wind up feeling about DeviceLoop? After we first discovered it, our tendency was to use it everywhere. We even used it to call a drawing routine that always drew in black and white, no matter what the bit depth. We later stripped this use out of the interface because it offered no advantage and added extra code.

One concern we had was that performance would degrade to an intolerable level as we added objects to be drawn. To see what would happen, the mischievous test engineer for our project devised a case with 99 separate TArt objects in the same window. Predictably, the 99 objects weren't displayed all at once. While you can expect some lag between the appearance of first object in a window and the last, however, the drawing time when you use DeviceLoop is really very short, well within user tolerance.

All in all, our design team was very pleased with DeviceLoop. We were glad to have found such an easy way to solve the problem of displaying interface objects on monitors of different bit depths. The interface designer got the look she wanted, and we were able to accomplish the job with a minimum of hassle and a minimum of code. This was one challenge that left everyone happy.

THANKS TO OUR TECHNICAL REVIEWERS

Edgar Lee and Brigham Stevens. Special thanks to Pat Coleman, the Interface Designer on the project that inspired this article.•

MACINTOSH Q & A

Q *Our program has a problem with filenames that start with a period. During an Open call, if the filename starts with a period, the Open code calls the Device Manager (for drivers and DAs) instead of the File Manager. However, we've seen other applications that can successfully open these files. What's the secret? How do we open files that otherwise look (from the name) like drivers?*

A The Open trap is shared between the Device Manager and the File Manager. When Open is called, it checks first to see whether you're trying to open a driver. Driver names always start with a period. If you can, avoid using filenames that begin with a period. Macintosh Technical Note "HFS Elucidations" (formerly #102) discusses this conflict. The secret to opening those files is using the new Open Data Fork functions available with System 7 — FSpOpenDF, HOpenDF, and PBHOpenDF. These functions bypass the driver name check and go right to the File Manager. Here's the code we use to open a file:

```
err := HOpenDF(vRefNum, dirID, fileName, permission, refNum);
IF (err = paramErr) THEN         {HOpenDF call isn't available}
   err := HOpen(vRefNum, dirID, fileName, permission, refNum);
                                 {try again with old HOpen call}
```

Try this and your problem should go away under System 7. The code retries with the regular Open call (which uses the same input parameters), so this code can be used in programs that run under both System 6 and System 7.

Q *In System 7, the memory allocated for INITs by the 'sysz' resource mechanism seems to be limited to about 16 MB on our 32 MB Macintosh IIfx. For 'sysz' values up to 15 MB it works great, but it seems the system heap can't grow beyond 16 MB. Is there some reason for this?*

A The system heap size at startup is limited to approximately half the size of total RAM. This is because the early startup code places the stack and some globals in the middle of RAM so that the system heap can grow up from below while BufPtr is lowered from above. This is basically the situation until an application is launched. Things are eventually rearranged so that the system heap will have more room to grow, but this doesn't happen until the Process Manager is launched, after INIT time. This limitation would mean that you could size your heap until it reached nearly (but not quite) half the size of RAM. We suggest that you attempt to allocate some of your RAM later, after the Process Manager starts up; at that point, the system heap should be somewhat less limited.

Q *The Macintosh Technical Note "Setting ioNamePtr in File Manager Calls" (formerly #179) says that ioNamePtr needs to point either to nil or to storage for a Str255. This*

Kudos to our readers who care enough to ask us terrific and well thought-out questions. The answers are supplied by our technical gurus in Apple's Developer Support Center; our thanks to all. Special thanks to Pete ("Luke") Alexander, Mark Baumwell, Joel Cannon, Matt Deatherage, Tim Dierks, Marcie ("M. G.") Griffin, Bill Guschwan, C. K. Haun, Dave Hersey, Dennis Hescox, Rich Kubota, Jim Luther, Joseph Maurer, Guillermo Ortiz, Kent Sandvik, Brigham Stevens, and Dan Strnad for the material in this Q&A column.•

contradicts the Technical Note "Searching Volumes — Solutions and Problems" (formerly #68), which gives an example of a recursive indexed search using PBGetCatInfo. The example uses a Str63. Which Technical Note is correct?

A To be generically correct, ioNamePtr should point to a Str255. However, in the case of PBGetCatInfo and other calls that return a filename (or a directory name), a Str63 is sufficient. The reasons are tied to the history of the Macintosh file system.

MFS, the original Macintosh file system, supported filename lengths of up to 255 characters. However, the Finder on those systems supported filename lengths up to only 63 characters and, in fact, developers were warned to limit filename lengths to fewer than 64 characters (see page II-81 of *Inside Macintosh* Volume II).

HFS, the hierarchical file system (in every Macintosh ROM since the Macintosh Plus), further limited filename lengths to 31 characters. If you mount an MFS disk while running HFS, the old MFS code is called to handle the operation. So, the file system can still create and use files with long filenames on MFS volumes.

When the System 7 file system was being designed, Engineering had to decide what size string to use in FSSpec records. The decision was to use a Str63 instead of a Str31 to be able to support long MFS filenames, and to use a Str63 instead of a Str255 because there were probably very few filenames with over 63 characters (remember, the old Finder limited filenames to 63 characters). Using a Str63 instead of a Str255 saves 192 bytes per FSSpec record.

So, we recommend that you use at least a Str63 for filenames, as in "Searching Volumes — Solutions and Problems." If you need to manipulate the filename in any way after you've gotten the name — for example, to concatenate it with another string — you might want to use a Str255.

Note: Even though the System 7 file system supports filenames longer than 31 characters on MFS volumes, the System 7 Finder does not. In fact, the System 7 Finder currently crashes if you try to open an MFS volume (that is, open the volume window) that has files with names longer than 31 characters.

Q *I'm trying to use the Macintosh Time Manager to calculate elapsed times, but when I increase the delay time from $4FFFFFF to $5FFFFFF I get incorrect results. Why is this happening?*

A There seems to be an undocumented limitation of the Time Manager: it can't keep track of times longer than about a day, so it replaces them with the

maximum time it supports. For Time Manager tasks, this isn't crippling; the task simply executes earlier than expected. When used for elapsed-time calculations, however, it's a bad thing; the Time Manager installs the task with the smaller time, and when you remove it, you see a smaller than expected remaining time. This makes it appear as if a large period of time has passed.

The value at which the Time Manager trims is approximately $53A8FE5. The reason for this strange value is somewhat complex. The Time Manager uses a VIA timer to do its measurement. This timer runs at 783360 Hz, giving it a resolution of about 1.276 microseconds. However, the Macintosh could never actually provide this kind of accuracy, given its latencies and overhead. Also, this frequency would have given a 32-bit counter a range of only about 91 minutes. Therefore, the Time Manager actually throws away the low four bits of this counter, keeping a 32-bit counter with a resolution of 20.425 microseconds and a range of 24 hours, 22 minutes. This time is a lot larger than the maximum number of microseconds that can be measured, but is equal to 87,724,005 milliseconds, which is (ta-dahh!) $53A8FE5. This is why you were overflowing the Time Manager's internal counter, causing your task to be clipped.

All should work well if you use times less than 24 hours. If you need to measure durations for times exceeding the Time Manager's limits, you can use a fixed-frequency task that executes every hour and increments an hour counter. To determine the fractional hours component of the time, you'd remove the task to determine how much longer till the next hour.

Q *When a picture that contains a pixMap is spooled into a window, how and when is the depth of the pixMap in the picture converted to the depth of the screens the window is on?*

A When a picture is spooled in, if QuickDraw encounters any bitmap opcode, it allocates a pixMap of the same depth as the data associated with the bitmap opcode, expands the data into the temporary pixMap, and then calls StdBits. StdBits is what triggers the depth and color conversions as demanded by the color environment (depth, color table, B&W settings) of the devices the target port may span (as when a window crosses two or more screens).

If there's not enough memory in the application heap or in the temporary memory pool, QuickDraw bands the image down to one scan line and calls StdBits for each of these bands. Note that if you're providing your own bitsProc, QuickDraw will call it instead of StdBits.

This process is the same when the picture is in memory, with the obvious exception that all the picture data is present; the color mapping occurs when StdBits does its stuff.

Q *How do I get the pixel depth of the QuickTime video media for a given track?*

A To find the video media pixel depth, you'll need to retrieve the media's image description handle. You can use GetMediaSampleDescription to get it, but this routine needs both the video media and the track's index number. It's not obvious, but a media's type is identified by its media handler's type. Thus, you can walk through a movie's tracks by using its indexes until you find video media, at which point you have both the track index and video media.

The following sample code does the trick:

```
#include <QuickDraw.h>
#include <Movies.h>
#include <ImageCompression.h>

Media GetFirstVideoMedia(Movie coolMovie, long *trackIndex)
{
   Track    coolTrack = nil;
   Media    coolMedia = nil;
   long     numTracks;
   OSType   mediaType;
   numTracks = GetMovieTrackCount(coolMovie);
   for (*trackIndex=1; *trackIndex<=numTracks; (*trackIndex)++) {
      coolTrack = GetMovieIndTrack(coolMovie, *trackIndex);
      if (coolTrack) coolMedia = GetTrackMedia(coolTrack);
      if (coolMedia) GetMediaHandlerDescription(coolMedia,
         &mediaType, nil, nil);
      if (mediaType = VideoMediaType) return coolMedia;
   }
   *trackIndex = 0;  // trackIndex can't be 0
   return nil;       // went through all tracks and no video
}

short GetFirstVideoTrackPixelDepth(Movie coolMovie)
{
   SampleDescriptionHandle imageDescH =
      (SampleDescriptionHandle)NewHandle(sizeof(Handle));
   long     trackIndex = 0;
   Media    coolMedia = nil;
   coolMedia = GetFirstVideoMedia(coolMovie, &trackIndex);
   if (!trackIndex || !coolMedia) return -1;  // we need both
   GetMediaSampleDescription(coolMedia, trackIndex, imageDescH);
   return (*(ImageDescriptionHandle)imageDescH)->depth;
}
```

Q *What's the difference between ignorance and apathy?*

A We don't know and we don't care.

Q *Could you tell me what the "printer driver is MultiFinder compatible" bit is used for?*

A The "printer driver is MultiFinder compatible" bit provides two features. First, it allows the printer driver resource file to be opened by multiple clients. This was obviously needed to support multiple applications printing simultaneously under MultiFinder. The other feature provided by the flag is the loading of PDEFs into the system heap rather than the application heap (which is where they go under the Finder).

The MultiFinder-compatible bit has a major limitation: if your driver has this flag set, you aren't allowed to add or resize resources, or do anything else that would cause the RAM-resident resource map to change. Although MultiFinder lets multiple applications open the printer resource file at the same time, it has no control over the resource map that gets loaded by the Resource Manager when the file is opened. Because of this, each client gets its own personal copy of the resource map. When these clients get done with the file, they write the resource map back to the file (via UpdateResFile). Obviously, if the resources have changed in any way, the last client to call UpdateResFile is the only one whose changes will be recorded. This is a "thrill seeker" method of handling the printer driver resource files, but since none of the Apple printer drivers currently add or resize resources, it made sense.

So the bottom line here is that if you want your drivers to be compatible under MultiFinder, you'll have to implement a scheme that doesn't require adding or resizing resources. It's OK to change the data in a resource, as long as you don't change its size. Changing the data won't cause changes to the resource map, so each client will still have accurate copies of the map.

Here's what would happen to your printer driver's resources under the Finder and MultiFinder when the MultiFinder-compatible bit is set:

- Under the Finder in system software version 6.0.x: All resources are loaded into the application heap — regardless of the resource attribute's bit setting. If the resource has the "load into the system heap" bit set, it will still be loaded into the application heap under the Finder. This makes sense in the Finder world because the application heap will usually have more room than the system heap.
- Under MultiFinder in System 6 or System 7: All the printer driver's resources *will* be loaded into the system heap. This is true whether the "load into the system heap" bit is set or not.

Why does the resource loading occur this way, even when the resource's "load into the system heap" bit is set? Patches to the GetResource trap load all your printer driver's resources into the system heap when the MultiFinder-compatible bit is set under MultiFinder, and into the application heap under the Finder (as described above), which is why you can't override this behavior.

By the way, you should be aware of the SetPDiMC MPW tool, which is available on the *Developer CD Series* disc. It will automatically set the MultiFinder-compatible bit for you when you build your printer driver.

Q *If I call FSWrite and attempt to write more than space allows, what happens? Of course I get a Disk Full error, but does FSWrite write as much as possible before quitting, and then return the number of bytes written in the count parameter?*

A In the current implementation of the file system, writes to local volumes owned by the file system are an all-or-nothing deal. If the space for a write can't be allocated, the write call fails and no bytes are written.

However, do *not* depend on that, because the Macintosh file system doesn't control all volumes that might be mounted. Today, Apple ships four external file systems: CD-ROM, AppleShare, ProDOS File System (for Apple II ProDOS volumes), and PC Exchange (for MS-DOS volumes). Various third parties have written other external file systems. The way they react to error conditions may not be the same as local volumes controlled entirely by the file system.

To make your application always work correctly, you should check for errors and handle them appropriately. If you get a dskFulErr, you should assume that if any information was written to the file, it wasn't written correctly. You should either reset the file's EOF to its previous position (if you're appending to an existing file) or delete the file (if you had just created the file and were writing to it for the first time).

Q *How can I mount a volume without using aliases? I get the mounting information, then attempt to mount the volume. However, the PBVolumeMount call returns an error code.*

A The PBGetVolMountInfo, PBGetVolMountInfoSize, and PBVolumeMount functions are currently handled by only the AppleShare external file system (part of the AppleShare Chooser extension). Those functions are available on AppleShare volumes when the AppleShare Chooser extension is version 7.0 (system software versions 7.0 and 7.0.1), version 3.0 (AppleShare 3.0), or version 7.1 (System 7.1). The AppleShare Chooser extension version 3.0 can be installed on System 6 systems, and then the PBGetVolMountInfo, PBGetVolMountInfoSize, and PBVolumeMount functions can be used in

System 6. Other file systems may support these functions in the future. The paramErr error code is returned when these functions aren't available on a particular volume.

Q *I need to prevent users from copying my application off a volume. Is there a new equivalent of the old Bozo bit?*

A The Bozo or NoCopy bit was bit 11 in the fdFlags word of the FInfo record. As noted in the Macintosh Technical Note "Finder Flags" (formerly #40), this bit hasn't been used for that purpose since System 5. In fact, System 7 reused bit 11 for the isStationery bit. (See *Inside Macintosh* Volume VI, pages 9-36 and 9-37, for the current list of Finder flag bits.)

There isn't an equivalent of the Bozo bit. However, the System 7 Finder won't copy files that have the copy-protect bit (bit 6) set in the ioFlAttrib field returned by the PBGetCatInfo function. However, the bits in the ioFlAttrib field can't be changed with the PBSetCatInfo function. Instead, they're used to report the state of things set by other parts of the file system.

The copy-protect bit is set by the AppleShare external file system when it finds that a file's copy-protect bit is returned by an AppleTalk Filing Protocol file server. The AppleShare external file system is the only file system we know of that sets the copy-protect bit. There's no way to make the local file system set the copy-protect bit for volumes it controls.

Q *Are there any tricks that might speed up reading and writing large files to disk? We're using standard C calls (fread and fwrite) for this purpose since our file I/O calls need to be platform-independent. Are there any low-level File Manager calls that might speed up the file I/O?*

A The C fread and fwrite functions are slower than File Manager calls because the standard C library adds another layer of overhead to file I/O. In addition, unless you turn buffering off, all file I/O is double-buffered when you use the standard C library functions. That is, fread reads from a RAM buffer in which the lower-level C library code has buffered data read from a disk file; fwrite writes data into a RAM buffer and the lower-level C library code writes from that buffer into a disk file.

For the highest file I/O throughput, and for maximum flexibility and functionality on the Macintosh, you should use the File Manager for all file access. The low-level File Manager calls (the PBxxx or PBHxxx calls) have the least overhead and give you the most control. If you use the File Manager's Read (FSRead or PBRead) and Write (FSWrite or PBWrite) calls, you'll achieve maximum throughput by reading or writing your data in the largest size

possible (for example, if you need to write 10,000 bytes, you can write them with one Write call).

If you must use the standard C library, you may want to adjust the size of the file I/O buffer used by the library for your particular purposes. You can adjust the size of the file I/O buffer using MPW C's setvbuf function. If you do nothing, you'll get a default buffer with a size of 1024 (1K).

MPW C's setvbuf size parameter is treated internally as an unsigned short. This means that the largest value acceptable to setvbuf for its size parameter is 65535. Larger values will be treated modulo this number. If you set the buffer size to 0, I/O is unbuffered. You can turn off buffering like this:

```
setbuf(stream, NULL);  // turn off buffering
```

or like this:

```
setvbuf(stream, NULL, _IONBF, 0);  // turn off buffering
```

Here are some general rules to follow to determine the size of the file I/O buffer you should use:

- If the file is small (less than 10K), you should probably use the default buffer.
- If the file is large (greater than 10K) but you write to it from your program in small pieces, buffering may help cut down the number of disk accesses. You may want to change the buffer size to around 10K. You can experiment to see whether other values provide better results for you. You'll probably find some point where the overhead of double-buffering is more than the overhead of disk accesses — that's when you should turn buffering off.
- If the file is large (greater than 10K) and you write to it in large pieces or write to it with one Write call, you should turn buffering off.

Q *If you were omnipotent and you had a round knob that controls the value of π, what would happen to the knob as you turned it?*

A Although unsure, we believe that the number of fingers on your hand would change.

Q *I'm porting C code from a UNIX® platform to the Macintosh. The code uses stdlib and stdio calls such as calloc, realloc, malloc, free, memcmp, memcpy, memset, strtod, strcat, strchr, strcpy, strlen, strncat, strncpy, strrchr, fopen, fclose, fwrite, and fread. For the most part, I've always avoided these calls on the Macintosh since the Toolbox has equivalents. However, I'd like to know whether there are any ramifications if I use these*

calls for porting compatibility. The only issues I can identify are (1) StdCLib.o, which defines these calls, uses globals and therefore will prevent me from using the code in standalone code segments, and (2) I'll lose some file information such as type and creator. Are there any other issues that I should be aware of?

A There are various difficulties or "gotchas" associated with use of these calls on the Macintosh, which generally keep them from being used in commercial development. However, being able to cross-compile code is very useful, so people like to use the calls for portability reasons despite their drawbacks.

The memory allocation calls (such as malloc, calloc, and realloc) all allocate pointer-based blocks. This works but can cause memory fragmentation and inefficient usage compared to the handle-based system usually used on the Macintosh. Also, MPW's implementation of these calls doesn't return memory to the Macintosh pool; when you allocate a block with malloc, the routine gets a larger block from the Macintosh with NewPtr, which it then subdivides into several smaller blocks to satisfy allocation requests. However, if the program then frees all the allocations made from this Macintosh pointer block, the library routine won't notice and dispose of it. Although the memory remains available for reuse by the standard C allocation routines, it has been lost to the Macintosh. For details, see the Q&A about using calloc and NewPtr in the same program in *develop* Issue 12 and the Macintosh Technical Note "A/UX Q&As."

The file manipulation calls suffer somewhat merely because they don't fit well into the Macintosh file system. For example, if you want to select files with the Macintosh StandardGet dialog, you'll find that fopen doesn't accept the volume reference or directory ID returned; it accepts a pathname, making it difficult to specify files in various folders. Also, as you noted, you have no control over types or creators; you also can't easily associate resource forks with data forks or use a number of the more expressive Macintosh file system calls.

You can use all of the string-manipulation calls (such as strcpy and strlen) and simple memory-access calls (such as memcpy and memcmp) with impunity; fortunately, bytes is bytes. Note, however, that a large number of seemingly innocuous calls (such as atoi and many others) use globals, making them inappropriate for use in cases where globals wouldn't be available, such as in code resources.

Basically, the standard C calls do work but suffer from faults, primarily because they've been kind of wedged into a system in which they don't fit. While most are functional and compatible enough to be used in software safely, be aware of their drawbacks and limitations; the basic decision is whether you can provide the functionality you need with these calls and whether the extra work required to deal with them is more or less than the effort saved by avoiding wholesale modifications to the source being ported.

Q *How can I detect whether a font suitcase is corrupted when it's opened and whether any of the fonts in it are corrupted before any of the fonts are used? I know that the Finder is able to do this, and I was wondering whether Apple gives out this information. My program will run only under System 7, if that helps.*

A The Finder and font architecture on the Macintosh are living things; the definition of what is and isn't a damaged suitcase can change from release to release of system software. However, any of the following conditions makes System 7 report the suitcase as "damaged":

- More than eight FONDs reference the same font.
- The Finder can't create a new standalone icon object for each font in the suitcase. The usual cause of this is that two FONDs have the same name for the first 31 characters, and the Finder thinks there's already an icon in that window with the same name. (Two icons in the same directory with the same name is a sign of damage.)
- There isn't at least one font association table entry, or the table goes past the logical end of the resource.
- The first resource name in the map is zero-length. (This is a test for some older third-party corrupted suitcases.)
- The FOND has no name.
- The FOND doesn't have a valid character range; ffFirstChar has to be less than ffLastChar, unless it's a "dummy" FOND (created on the fly for old standalone FONTs, in which case ffLastChar is 0).
- All the font association table entries aren't in ascending point size order.
- Two font association table entries reference exactly the same point size and style.
- The offsets to the width table, kerning table, and style mapping table are invalid and nonzero.
- The font ID is 0 and it's not the system font.

We can't promise that this list is complete, but it does contain most conditions for which the Finder would report a suitcase as damaged.

Q *We'd like to maintain only one version of our globally distributed application, which would adapt to the language in use by changing DITL resource text items and menu titles and items. Does the Macintosh Operating System support this?*

A Currently the Macintosh Operating System doesn't inherently support localized resources for several languages, or choose the right language according to the

localized version of the system. However, your approach of including all localized text items in the same application is absolutely feasible. Just include an option to let the user select the language — somewhere in Preferences, if not in a dedicated "Languages" menu — and design a numbering scheme for the resource IDs such that the resources to be loaded can be determined from the language code.

It's better to let the user choose the language, rather than derive it from the system. This provides for a choice in case the user lives in a multilingual region, or in case your application doesn't include translations for the language of the user's system.

Because menus, windows, and dialogs are displayed with the system font, this approach works only for languages supported by the system script.

Q *My installer creates a folder on a user's hard disk and copies the necessary files into it. My final action atom moves the folder onto the desktop and sets its size and location. I'd also like to be able to open the folder. I call PBGetCatInfoSync to get the data into a CInfoPBRec record. Where is the state of a folder (open/closed) stored, and can I set one of the parameters in the CInfoPBRec and then call PBSetCatInfoSync to solidify the change? Using the installer to copy an open folder to the user's drive is unacceptable because of the size and nature of the program I'm installing.*

A There's no solution for System 6; the Finder data structures are private, and there's no call to open a folder. In System 7, you can send the Finder an Open Selection Apple event. This is described in a HyperCard stack called FinderEvents on the *Developer CD Series* disc. The stack also contains the source code for the XCMD used to demonstrate the Finder events. There's another sample that you should see as well: SendFinderOpen in the Snippets folder.

Q *We're having problems with color patterns using the LaserWriter driver version 7.1.2. If we create a 'ppat' resource in ResEdit (32 x 32 bits, in this case) and then do a FillCRect to the port returned by PrOpenDoc (with color set so that it's a cGrafPort) with the pattern loaded by GetPixPat, we get a weird pattern. Doing the same to an off-screen GWorld and using CopyBits to copy to the printer port works fine, if a little slowly. Are we missing something here?*

A You need to use the FillCRect call off-screen rather than directly into the printer port, for at least two reasons. First, the LaserWriter driver doesn't support filling objects with anything but black-and-white patterns because it uses the PostScript halftone screen functions to draw patterns. Second, the LaserWriter driver doesn't understand (or handle) pixPats. Therefore, your only recourse is the one you discovered — to copy to and from GWorlds.

Unfortunately, FillCRect doesn't work with the LaserWriter drivers through version 7.2. After version 7.2 this probably won't be a problem.

Q *Do NumToString and StringToNum work correctly regardless of the script chosen as the system script? When I attempt to use SANE to convert non-Roman digits from a dialog box editText item, SANE doesn't seem to like it.*

A SANE expects all digits to be in the range ASCII $30–$39, with $2D as a negative indicator. These ASCII values can be generated from any international script by using the Macintosh numeric keypad. The symbols 0 through 9 are internationally recognized as numeric values.

There are many additional ways to represent numbers on the Macintosh, including words (one, two, uno, dos), notations (dozen, hundred, million), ordinals (first, second, third), Roman numerals (I, II, III), symbols (π, e, i), and hexadecimal ($FF). Many languages have alternative numbering systems and special characters that represent numbers. In Symbol and double-byte fonts, there are special characters representing fractions (1/2, 1/4), superscripts, subscripts, numbers within circles, and so on.

While it would be nice to have routines that convert between ASCII numbers and alternatives such as longhand numbers (used when writing checks), Roman numerals (used for copyright year in movie credits), or local number systems (for formal documents), no such routines exist in the Macintosh Toolbox today. It would be possible but difficult for an application to custom-process numbers for each language and script. The *Unicode Standard Reference,* Volume 1, lists hundreds of different kinds of numbers — and they're not all base 10.

Scripts that have alternative number character sets always support the universal single-byte ASCII digits as well. When a script has alternative numeric characters, the user generally types script-dependent numeric characters from the top row of the keyboard and the single-byte ASCII digits from the numeric keypad.

Although it doesn't translate the digits themselves, the Script Manager offers support for formatting a number into a local form. For example, Europeans often use a comma as a decimal point and a period as a thousands marker. Most countries have unique currency symbols. There are many different ways to represent numerical values for things such as date, time, and money. This kind of formatting information is in the international resources.

One way to do data validation is to use CharType and check for numeric characters. We can't guarantee that this has been implemented for all scripts, but it is correct for Roman and Japanese.

NumToString and StringToNum don't deal with international formats. Use the Script Manager routines Str2Format and Format2Str to get the text into a numerical form that SANE can deal with. See *Inside Macintosh* Volume VI, page 14-49, for details.

Q *I'm attempting to determine whether a debugger is installed, and if so, to find a THz pointer to its heap zone. Is this possible?*

A The MacsBug debugger is loaded into high memory above the value found in the global variable BufPtr ($10C). Since it's loaded into the memory that's not managed by the Memory Manager, it's not in a heap. The global variable MacJmp ($120) points to the debugger's entry point.

There's also a flags byte in low memory that contains the following information:

Bit 7	Set if debugger is running.
Bit 6	Set if debugger can handle system errors.
Bit 5	Set if debugger is installed.
Bit 4	Set if debugger can support the Discipline utility.

The flags byte may be in one of two places: the high-order byte of the long word at MacJmp, or the address $BFF. When MacsBug is loaded, it examines the value at address $BFF. If the value at $BFF is $FF, the system is using the 24-bit Memory Manager and the flags byte will be the high-order byte of the long word at MacJmp. If the value at $BFF isn't $FF, the system is using the 32-bit Memory Manager and the flags byte will be at address $BFF.

For information on debuggers other than MacsBug, you'll need to contact the publishers of those products.

Q *We need to localize our application for several international markets. Do you have any special tools or recommendations for us?*

A You can use a System 7.1 tool called AppleGlot (on the *Developer CD Series* disc) to localize text in your application. Once a file has been localized the first time, the tool can compare versions and copy over everything that has stayed the same (usually 99%) so that it can focus on the text that's different. It also creates a nice audit trail and is pretty easy to use. It should save you a lot of time.

To take full advantage of this tool, you need common code for all localized versions, which is what you're planning to do to avoid the mess of having multiple sources. Occasionally, your application might have features that make sense only on a particular script system; in that case, you can check for that

configuration and enable those routines when appropriate. Once you have common source and tools that help localize your application, you can add auxiliary resources for various languages.

If you have only a small amount of text in your application, it makes sense to bundle everything together in one worldwide product. Apple's TrueType fonts, for example, have internal name tables with names and information such as copyright strings in about a dozen languages. Each string is tagged with a platform, script, and language. But if you have a fair amount of textual resources, it might make more sense to have optional files and resources that can be installed as needed.

Unless you intend to support every script and language, you'll probably want to have a set of resources for unavailable languages. You can pick whatever language you want for this other set (English is popular), but the trick is to use only 7-bit ASCII characters. All script systems use the same character codes for the range $00–$7F, which match ASCII. It's the 8-bit characters that differ radically. This means that text that includes characters like …, ™, ©, and • will not display properly on non-Roman script systems. Just substitute text such as . . ., tm, (c), and * for them. You can decide what's appropriate and necessary.

Another thing to consider is checking for and supporting secondary script systems in your application. The Macintosh Toolbox doesn't fully support secondary scripts such as Japanese menus on an English system, but your application can support secondary script data even with the current Toolbox limitations, by using styled text commands.

Q *We would like to use the "dogcow" icon in our Page Setup dialog. Is the dogcow trademarked, and are there any restrictions on using this icon in our software?*

A Yes, the dogcow logo (along with its call, "Moof!") is a trademark of Apple and is proprietary. The dogcow appears on Apple's *Developer CD Series* disc and in other material. Apple has a pending U.S. registration on it. Accordingly, it's not available to third-party developers as an icon or file symbol.

Q *Where in the world does the dogcow come from?*

A Some people say that the dogcow hails from the sunny shores of the Middle of Nowhere. This location in the south Atlantic can be found in the Map control panel; simply type "Middle of Nowhere" and click Find. (For a small fee, these same people will tell you where they last saw Elvis.)

Have more questions? Need more answers? Take a look at the Q&A Technical Notes on the *Developer CD Series* disc and the Dev Tech Answers library on AppleLink.•

KON & BAL'S PUZZLE PAGE

BOOTING BLUES

After answering a question they tantalizingly left hanging in Issue 12 of develop, *Konstantin Othmer (KON) and Bruce Leak (BAL) go on to present yet another programming puzzle in the form of a dialog between them. The dialog gives clues to help you. Keep guessing until you're done; your score is the number to the left of the clue that gave you the correct answer. These problems are supposed to be tough. If you don't get a high score, at least you'll learn interesting Macintosh trivia.*

KONSTANTIN OTHMER AND BRUCE LEAK

KON So have you figured out what's the fewest keystrokes needed to do an ExitToShell from MicroBug?

BAL Aren't you at least going to say "Good morning"?

KON Good morning, BAL. Well . . .?

BAL Well, you can do it by calling through the trap table, or by doing an SM PC A9F4 and then G.

KON Good idea, but that definitely won't work if you're in ROM, and it could really hose you if you're somewhere in the system and you modify RAM-based system code.

BAL How about if you jump to the bus error vector by doing a PC=@8, and then G?

KON MicroBug, not MacsBug! It can't do indirection. What do you think the "Micro" stands for?

BAL Fine. Manually set the PC to the address at 8.

KON Four keystrokes for the DM 8, eight keystrokes for the PC=XXXXX, and another one for the G. I'm not even counting all the return characters. But it works great.

BAL OK, OK. PC=-1, G. That will cause a bus error and MultiFinder will deal with it as long as MultiFinder is in a reasonable state at the time.

KONSTANTIN OTHMER AND BRUCE LEAK
After cashing in on their *develop* incentive points from the highly acclaimed Puzzle Page, KON and BAL have diversified their interests from programming and publishing to predicting the stock market. Although their attempts to corner the DRAM market backfired, they still have high hopes. KON says, "I can't believe we didn't do this before — talking to our brokers while we're programming." BAL adds, "Yeah. True multitasking, and we don't even need a preemptive, multithreaded OS."•

KON Come on BAL, that's six keystrokes. You can do better than that.

BAL OK. G -1 does the same thing in four keystrokes. Beat that!

KON Not bad — but I can crash that machine in three keystrokes. G -1 is the best you can do for a bus error, but you can get an address error in three strokes: G 1.

100 BAL Fine. If you think you're so smart, figure this next one out: We got our QuickTime 1.5 test disc back and I decided to try it out on System 6. I took a Macintosh si we used for development and downgraded it from System 7 to system software version 6.0.7. After I installed QuickTime 1.5 from our test disc, the machine crashed at boot time.

KON You crashed while QuickTime was loading, right?

BAL Your puzzle, KON.

KON Hmmm. So I reboot with the Control key down to get into MacsBug. Then I do a

```
atb HOpenResFile ';dm @(sp+2); g
```

and see what was the last resource file that was opened. That way I can find who's causing the crash.

BAL System 6, future boy.

KON OK, so I use

```
atb OpenResFile ';dm @sp; g
```

90 BAL Surprise, surprise — it's QuickTime. You could have just used the File dcmd after you crashed and noticed that the QuickTime file was still open.

KON So you have some other INIT in there that's got a conflict with QuickTime 1.5. Boot with the Shift key held down and throw out everything else.

BAL Take a hike. This is System 6. Your brain's gone soft.

KON Well, just use the skipInits macro.

80 BAL Sorry, I don't have that one.

KON D'oh! [Watch *The Simpsons* for correct pronunciation.] So I skip over all the INITs by making them fail using

```
atb GetIndResource @(sp+2) = 'INIT' ';sp=sp+6; @sp.l=0;
pc=pc+2; g
```

	BAL	Wow! How'd you whip that one up?
	KON	I just checked out page 326 of that excellent book, *Debugging Macintosh Software with MacsBug*. You know, it tells you all about how INITs work in both System 6 and System 7, it even . . .
75	BAL	Yeah, yeah, yeah. More shameless self-promotion. I have the Dimmer and Programmer's Key INITs, Apple CD-ROM stuff, AppleShare, and QuickTime, of course.
	KON	So throw all those INITs out of there except QuickTime, and go buy Delta Tao's WonderPrint since you need it anyway.
70	BAL	You still crash, same time, same place, same channel.
	KON	Try it without QuickTime. Maybe System 6 didn't get installed correctly.
65	BAL	Boots fine.
	KON	So either this copy of the INIT is bad or QuickTime 1.5 is lame under System 6. Try the INIT somewhere else.
60	BAL	I have a System 6 LC and a System 7 si, and that copy works fine on both, especially the new compact video compressor. Totally awesome. I can't believe we give it away for free. Windows: $99; QuickTime: free. Go figure.
	KON	So what's the difference between the two si's, other than one is running System 7? Does either have that FPU-NuBus slot adapter?
55	BAL	No slot adapter. Both have 4 megs.
	KON	Same video mode? Both booting in 24-bit mode?
50	BAL	Good question, but the System 7 version works fine in 24-bit mode, and the monitor depths are the same.
	KON	So maybe there's some kind of boot block problem, or some other conflict arising since the machine was downgraded. Convert the other si to System 6 and try it.
45	BAL	OK, works fine.
	KON	So what you're telling me is you take two si's, install System 6 on both from the same disks, neither has any external hardware or INITs, and when you install QuickTime, one crashes and the other doesn't.
	BAL	Your puzzle, KON.
	KON	Same system, same INITs, same hardware — there must be some state that's different. Something is funny about PRAM: the addressing mode, disk cache size, sound volume, default video mode, network settings, mouse speed. One of those things must be hosing us.

SCORING

75–100 Yeah, right. You still have that bridge for sale?
50–70 QuickTime team members and their immediate family are not eligible.
25–45 Better than KON! Write a *develop* article.
10–20 You win! Honest people like you are rare these days!•

40 BAL Well, which ones do you want me to change? I don't have all day.

KON Put them all back to their default state with Command-Option-RP (Reset Pram) at boot time.

35 BAL Fine. I do that. While booting it reboots, letting you know it cleared PRAM. Then one crashes and the other one doesn't.

KON I swap hard disks.

30 BAL The crash follows the hard disk.

KON So maybe there's something funny going on with MacsBug. Are the versions the same?

25 BAL The versions are the same, but one is black text on white and the other is blue text on black.

KON I get it. Just pull MacsBug out.

20 BAL Boots fine. OK. So what's the answer?

KON There's something going on with Debugger Prefs.

15 BAL How could they affect things at boot time?

KON The FirstTime macro or the EveryTime macro.

10 BAL I don't have either.

KON Dcmds get an initialization call and one of them is hosing the system. It's probably not System 6 friendly.

BAL OK. So how do you figure out which one is causing the problem?

KON Keep taking them out half at a time until you find the problem, or just look for the skanky ones Darin wrote.

BAL Yeah, there was a dcmd in Debugger Prefs that was used for debugging System 7 patches. QuickTime uses the same loading mechanism as some of the System 7 patches, and this dcmd overrides the loading mechanism for debugging purposes. So the dcmd assumed a System 7 runtime environment, which turned out to be a bad assumption.

KON Nasty.

BAL Yeah.

Thanks to Gary Davidian and scott douglass for reviewing this column.•

INDEX

A

ActivatePalette 91
"Adventures in Color Printing" (Hersey) 64–90
Alexander, Pete ("Luke") 52
aliases, Macintosh Q & A 109–110
Angus User Area, QuickTime and 38, 39
application heap, asynchronous routines and 29–30
'Asm ' resources, QuickTime and 37, 38, 39
assembly language, debugging and 58–59
asynchronous routines 5–30
"Asynchronous Routines on the Macintosh" (Luther) 5–30
A-traps, QuickTime and 34–51

B

banding, color printing and 73–76
BeginUpdate, DeviceLoop and 99, 100
black and white
  color printing and 82–85
  palettes and 93–96
Blat, debugging and 58–59
booting, KON & BAL puzzle 118–121
bottlenecks, QuickTime and 45–51
Boyd, Scott 10

C

C, Macintosh Q & A 110–112
call chaining, asynchronous routines and 18
cGrafPorts, color printing and 64–90
Chooser, QuickDraw GX and 53
'clok' component, QuickTime and 45
CloseResFile, debugging and 55
Color Adventures sample, color printing and 69
Color ImageWriter Adventures sample, color printing and 77, 78, 81
color patterns, Macintosh Q & A 114–115
color printing 64–90
Color QuickDraw
  color printing and 64–90
  DeviceLoop and 99
  palettes and 92
color tables
  color printing and 75
  palettes and 91–96
completion routines, asynchronous routines and 6, 10–15, 29
component bottlenecks, QuickTime and 47–51
Component Manager, QuickTime and 34–51
components, QuickTime and 34–51
CopyBits
  color printing and 67, 68, 74, 76–77, 79
  Macintosh Q & A 114–115
copying files, Macintosh Q & A 110
custom dialogs, QuickTime and 32
CustomGetFilePreview, QuickTime and 32, 41, 45

D

DAs, Macintosh Q & A 104
data reference limitations, QuickTime and 31
"Deadlock" (Sheridan) 7
Debugger routine
  asynchronous routines and 25
  QuickTime and 45

For a cumulative index to all issues of *develop*, see the *Developer CD Series* disc.•

debuggers
asynchronous routines and 25
Macintosh Q & A 116
debugging 54–60
asynchronous routines and 25–26
KON & BAL puzzle 118–121
QuickTime and 34–51
debugit, QuickTime and 37, 38
DebugStr, asynchronous routines and 25
Deferred Task Manager, asynchronous routines and 6
delay time, Macintosh Q & A 105–106
DelegateComponentCall, QuickTime and 49, 50
Dequeue, asynchronous routines and 15, 18
desk accessories (DAs), Macintosh Q & A 104
DeviceLoop 97–103
DeviceLoopInDrag, DeviceLoop and 100
"DeviceLoop Meets the Interface Designer" (Powers) 97–103
Device Manager
asynchronous routines and 7, 11, 12–15
Macintosh Q & A 104
DisposeHandle, debugging and 55, 56–58
Dispose Resource, debugging and 57–58
ditherCopy method, color printing and 82
dithering, color printing and 82–85
DITL resource, Macintosh Q & A 113–114
DoPPCInform, asynchronous routines and 12, 21, 23
DoPrint, color printing and 89
Double Trouble, debugging and 55–57
Draw, DeviceLoop and 102
DrawPicture
color printing and 73, 76, 78, 80
QuickTime and 46
DrawProc, DeviceLoop and 99, 100
drivers
color printing and 64–90
Macintosh Q & A 104
dynamic state, QuickTime and 43–45

E

elapsed times, Macintosh Q & A 105–106
EndComplete, asynchronous routines and 21, 23, 24
EndUpdate, DeviceLoop and 99, 100
Enqueue, asynchronous routines and 15, 18
ExitMovies, QuickTime and 31–32
extended parameter blocks, asynchronous routines and 18–20, 25–26, 28–29

F

file I/O, Macintosh Q & A 110–111
File Manager
asynchronous routines and 6, 7, 11, 12–15
Macintosh Q & A 104, 110–111
filenames, Macintosh Q & A 104
FillCRect, Macintosh Q & A 114–115
Final Adventure sample, color printing and 70, 89
Finder
DeviceLoop and 97
Macintosh Q & A 113
FindNextComponent, QuickTime and 42–43
fonts, Macintosh Q & A 113
font suitcases, Macintosh Q & A 113
ForeColor, color printing and 76
FrameOval, palettes and 91–92
FSWrite, Macintosh Q & A 109
"Function Results and Function Completion" (Boyd and Luther) 10

G

GDevices, color printing and 66–67, 74
Get1IndResource, QuickTime and 41, 42
Get1Resource, QuickTime and 41, 42
Get1xResource, QuickTime and 41
GetA0, debugging and 59
GetBestDPI, color printing and 70
GetCompressedPixMapInfo, QuickTime and 46
GetCTable, color printing and 77
GetImageRes, color printing and 69
GetMovieNextInterestingTime, QuickTime and 31
GetNewWindow, QuickTime and 42
GetPBPtr, asynchronous routines and 13
GetPixPat, Macintosh Q & A 114–115
GetResource, QuickTime and 41, 42, 43
GetRslData, color printing and 69–70

grafPort bottlenecks, QuickTime and 46
grafPorts, color printing and 64–90
"Graphical Truffles" (Lee and Tanaka) 91–96
Guschwan, Bill 34
GWorlds, color printing and 69, 73–77, 78, 79

H

Halftone Adventures sample, color printing and 82, 83, 84
halftoning, color printing and 82–85
HandlePPCErrors, asynchronous routines and 21, 24
hardware devices, QuickTime and 33
heap zone, Macintosh Q & A 116
Hersey, Dave 64
Huxham, Fred 54

I

Image Compression Manager, QuickTime and 35
Incompatibility Test, color printing and 73
InformComplete, asynchronous routines and 21–22
INITs, Macintosh Q & A 104
InitZone, debugging and 57
"Inside QuickTime and Component-Based Managers" (Guschwan) 34–51
installers, Macintosh Q & A 114
interface design, DeviceLoop and 97–103
internal routines, QuickTime and 39–43
interrupt handlers, asynchronous routines and 6
interrupt time, asynchronous routines and 6, 27–28

J

Johnson, Dave 61

K

"KON & BAL's Puzzle Page" (Othmer and Leak) 118–121

L

language, Johnson ponders 61–63
LaserWriter driver
    color printing and 75
    QuickDraw GX and 52–53
LaserWriter driver 7.1.2, Macintosh Q & A 114–115
LaserWriter Font Utility, QuickDraw GX and 52, 53
lazy person's halftone method, color printing and 83–85
Leak, Bruce 118
Lee, Edgar 91
List Manager, DeviceLoop and 97
localization, Macintosh Q & A 113–114, 116–117
Luther, Jim 5, 10

M

MacApp, Johnson ponders 61–63
"Macintosh Debugging: The Belly of the Beast Revisited" (Huxham and Marriott) 54–60
Macintosh Q & A 104–117
MacsBug
    KON & BAL puzzle 118–121
    QuickTime and 34–51
Marriott, Greg 54
matrix routines, QuickTime and 35
MaxApplZone, QuickTime and 33
memory
    debugging and 58
    Macintosh Q & A 104
Memory Manager, debugging and 55, 56, 57–58
MicroBug, KON & BAL puzzle 118–121
MMU, debugging and 58, 59
'moov' resource, QuickTime and 41, 42
mounting volumes, Macintosh Q & A 109–110
movie controller, QuickTime and 33
MoviePlayer, QuickTime and 41, 45, 50
movies
    debugging and 57
    Macintosh Q & A 107
    QuickTime and 31–33
Movie Toolbox, QuickTime and 32, 35, 41, 45
MPW C, debugging and 59
multiple monitors, DeviceLoop and 97–103
'mxbm' resources, QuickTime and 37, 38, 39

N

natural languages, Johnson ponders 61–63
NewMovie, QuickTime and 42
NewMovieFromDataFork, QuickTime and 42
NewMovieFromFile, QuickTime and 41, 42, 43, 45
NewMovieFromHandle, QuickTime and 42
NewMovieFromScrap, QuickTime and 42
NewTimeBase, QuickTime and 45
NewWindow, QuickTime and 42
NuMathComponent, QuickTime and 48–49
NumToString, Macintosh Q & A 115–116

O

off-screen GWorlds, Macintosh Q & A 114–115
OffsetRect, debugging and 58
“one size fits all” approach, color printing and 65–66
Open, Macintosh Q & A 104
OpenComponent, QuickTime and 43
OpenCPicture, color printing and 69, 78, 80
OpenDefaultComponent, QuickTime and 43
OpenPicture, color printing and 78
operating system queues, asynchronous routines and 15–18
Othmer, Konstantin 118

P

Palette Manager 91–96
palettes 91–96
'PAPA' resource, QuickDraw GX and 52–53
PBGetCatInfo, Macintosh Q & A 105
PBGetCatInfoSync, Macintosh Q & A 114
PBHOpen, asynchronous routines and 6
PBSetCatInfoSync, Macintosh Q & A 114
PBVolumeMount, Macintosh Q & A 109–110
PDEF resource, QuickDraw GX and 52–53
PICT resources, DeviceLoop and 99, 102
PICTs, color printing and 69, 73, 80, 81
pictures, Macintosh Q & A 106
Picture Utilities Package, QuickTime and 46
pixMaps
    color printing and 64–90
    Macintosh Q & A 106
'pltt' resource, palettes and 91–92
pmAnimated, palettes and 93
PmBackColor, palettes and 91, 93
pmCourteous, palettes and 91–92
pmExplicit, palettes and 93, 94, 95
PmForeColor, palettes and 91, 92, 93
pmTolerant, palettes and 92–93, 94
PollForCompletion, asynchronous routines and 8, 9
polling, asynchronous routines and 8–10, 29–30
PostScript, QuickDraw GX and 52
PostScriptHandleDemo snippet, QuickDraw GX and 52
'ppat' resource, Macintosh Q & A 114–115
PPCClose, asynchronous routines and 20
PPCEnd, asynchronous routines and 23, 24
PPCInform, asynchronous routines and 5, 8, 9, 11, 12, 20, 21–22, 23
PPCRead, asynchronous routines and 20, 21–22, 24
PPC Toolbox, asynchronous routines and 5, 7, 11–12, 13, 20
PPCWrite, asynchronous routines and 24
PrCloseDoc, color printing and 87
PreCompletion, asynchronous routines and 14
prerolling a movie, QuickTime and 32
PrGeneral, color printing and 69–70, 71, 74, 80
printer access protocol (PAP), QuickDraw GX and 52, 53
printer drivers, color printing and 64–90
“Print Hints” (Alexander) 52–53
printing, color printing 64–90
Printing Manager
    color printing and 67
    QuickDraw GX and 52, 53
PrJobDialog, color printing and 87
ProcessPPCData, asynchronous routines and 21, 23
programming languages, Johnson ponders 61–63
PrOpen, color printing and 86–87
PrOpenDoc
    color printing and 67, 86, 87
    Macintosh Q & A 114–115
PrPicFile, color printing and 87
Puzzle Page 118–121

Q

Q & A, Macintosh 104–117
QuickDraw
    palettes and 93
    QuickTime and 45, 46
QuickDraw GX, print hint 52–53
QuickTime 31–33, 34–51
    debugging and 57
    KON & BAL puzzle 118–121
    Macintosh Q & A 107

R

race conditions, asynchronous routines and 16
ReadComplete, asynchronous routines and 21–22
ReleaseResource, debugging and 57–58
ResEdit
    Macintosh Q & A 114–115
    palettes and 92

resolution, color printing and 69–72, 82
Resource Manager
debugging and 55, 57–58
QuickTime and 39–42, 43
RGBColor, palettes and 91, 93
RGBForeColor, palettes and 92, 93

S

SANE, Macintosh Q & A 115–116
scaling
color printing and 69–72
movies and 32
Separations Test, color printing and 80
SetA5, debugging and 59
SetGWorld, color printing and 73
SetPalette 91
SetPort, color printing and 68
Sheridan, Gordon 7
Smart Friends, debugging and 59–60
“Somewhere in QuickTime” (Wang) 31–33
stack space, asynchronous routines and 27–28
standard movie controller, QuickTime and 32
StdPix, QuickTime and 46
Str63, Macintosh Q & A 105
Str255, Macintosh Q & A 104–105
StringToNum, Macintosh Q & A 115–116
'STR ' resource, QuickDraw GX and 52–53
SyncWait, asynchronous routines and 26
System 6, KON & BAL puzzle 118–121
System 6.0.7, KON & BAL puzzle 118–121
System 7
KON & BAL puzzle 118–121
Macintosh Q & A 104, 113
UTILs and 53
system heap, Macintosh Q & A 104
system script, Macintosh Q & A 115–116
'sysz' resource, Macintosh Q & A 104

T

Tanaka, Forrest 91
TeachText, color printing and 73, 80
thing dcmd, QuickTime and 43, 50–51
'thng' resource, QuickTime and 42
TimeBase, QuickTime and 45
Time Manager
asynchronous routines and 6
Macintosh Q & A 105–106
TMON Pro, QuickTime and 34–51
trapping, QuickTime and 34–51
“true” halftone method, color printing and 83

U

UNIX, Macintosh Q & A 111–112
UTILs, print hint 53

V

VBL tasks, asynchronous routines and 6, 29–30
“Veteran Neophyte, The” (Johnson) 61–63
video media pixel depth, Macintosh Q & A 107

W, X, Y, Z

WaitNextEvent, asynchronous routines and 27, 29
Wang, John 31
Window Manager, DeviceLoop and 97, 98