develop

The Apple Technical Journal

Issue 16 December 1993

EDITORIAL STAFF

Editor-in-Cheek *Caroline Rose*

Technical Buckstopper *Dave Johnson*

Our Boss *Greg Joswiak*

His Boss *Dennis Matthews*

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

Managing Editor *Cynthia Jasper*

Contributing Editors *Lorraine Anderson, Philip Borenstein, Robin Cowan, Matt Deatherage, Toni Haskell, Judy Helfand, Rebecca Pepper*

Indexer *Marc Savage*

Special thanks to Smart Friend Dean Yu for his help during Dave Johnson's sabbatical.

ART & PRODUCTION

Production/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*

Online Production *Cassi Carpenter*

develop, The Apple Technical Journal, a quarterly publication of Apple Computer's Developer Press group, is published in March, June, September, and December.

The cover. Mark Jenkins of Rucker Huggins Design created this cover using Adobe Photoshop, Adobe Illustrator, Fractal Design Painter, and a Macintosh Quadra 950. He looks forward to making the leap himself to Macintosh on PowerPC.

This issue's CD. The *develop Bookmark* CD (or the *Developer CD Series* disc, Reference Library edition) for December 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. Note that some software and documentation referred to as being on this issue's CD may be located on the Tool Chest edition rather than the Reference Library edition of the *Developer CD Series* disc.

CAROLINE ROSE

Dear Readers,

Nearly two years ago I was asked to join a team of Apple developer support people who would meet monthly to discuss a forthcoming Macintosh model that would be based on the new PowerPC (RISC) processor. Our goal was to ensure that developers receive the support they need to get started on this new computer, due to be introduced in 1994.

It seemed a bit premature to me back then, but the time went faster than anyone would have imagined. Now I'm happy to finally have something to show for it in *develop*: an article to help you make the transition to this new stage in the life of the Macintosh. We hope to tear the engineers away from their programming long enough to be able to have a steady stream of articles on this subject in future issues.

The PowerPC processor–based Macintosh will be formally introduced roughly a decade after the introduction of the first Macintosh computer, which of course makes me wax nostalgic about what I was doing then at Apple: furiously finishing up *Inside Macintosh* on an Apple III, handing off chapters to Louella Pizzuti (later the founder of *develop*) to format them on the Macintosh in MacWrite®. But most of all, I remember looking forward to what I *knew* would be worldwide acceptance of — no, excitement about! — our new computer.

The world is different now, as am I (we've both changed a lot in ten years), and I'm shorter on starry-eyed wonder and exclamation points than I was back then. But to quote an old TV personality (trivia question: who?): "I think you're gonna like this one." Talk about time flying by, this thing is *fast*. And it's still a Macintosh (which I'll always have a warm spot for in my heart no matter where I roam). It's a horse of a different color, but it's still a horse: a sleek, beautiful racehorse that should make us all winners. At least that's what I'm betting.

Caroline Rose
Editor

CAROLINE ROSE (AppleLink CROSE) got into technical writing because of a Math degree she didn't know what to do with. It landed her a job at Tymshare, where she wrote her first manual in pencil on paper, from which someone typed it on an IBM Selectric typewriter. But enough ancient history. Caroline's career at Apple began with *Inside Macintosh* and almost ended when she left to join NeXT, but she was smart enough to come back after five years and become the editor of *develop*. After putting each issue of *develop* to bed, Caroline likes to take off to someplace where she can forget about computers altogether. After this issue, she'll fulfill a dream she's had since the first time she looked at a globe as a child: she'll visit the Caroline islands. She'll stay with friends who live on a sailboat in primitive style (except for their computers!).•

LETTERS

NEW CD & PACKAGING: GOOD

I just received *develop* Issue 14 in the mail (a little late, but worth the wait!). As usual, the articles are enlightening and entertaining. The Bookmark CD is an excellent idea as well (although the former student in me doesn't like the idea of missing out on system software). I enjoyed opening up the journal to find my CD in a case that could not be broken, even by the worst of mailmen! And, as always, the CD contains a great wealth of information. I would even go so far as to say I'd pay more than the price of a yearly subscription with or without the CD; it's really worth it! So, I'd like to congratulate you and everyone else on the *develop* staff (as well as all the contributors) for creating an amazing journal for me (and I guess other developers!) every three months.

Keep up the good work, as I look forward to future issues. (I just renewed my subscription.)

—David A. denBoer

Thanks for the good words about develop *and especially about its CD. We've received a number of letters from developers who are less than thrilled about our recent changes (see below). It's wonderful to hear from a happy subscriber.*

—Caroline Rose

NEW CD: BAD

I subscribed to *develop* after reading the *MacWeek* article that pointed out what a good deal it was, especially with respect to the *Developer CD Series*. APDA told me on the phone that my subscription would start in a few weeks, and promptly debited by VISA account.

After a few weeks had gone by, I phoned ADPA to ask why I hadn't received anything. I was told that they had been swamped with orders as a result of the *MacWeek* article.

Now I find out that there's been a "change" to *develop*'s CD. What bothers me is the apparent motivation behind all this. You knew that you were swamped with orders, which were generated by a specific *MacWeek* article. What did this article say? Simply that readers should subscribe to *develop* to get the system software. So what did you do? Take away the software! Didn't it occur to anyone making this decision that loyal customers might be angered?

I, for one, am not a happy camper.

—Ken Ribet

Regarding the CD change: We basically took only Inside Macintosh and system software off the CD. We have since restored Inside Macintosh; that decision was made a bit hastily. But I think the decision to discontinue essentially giving away the system software was a sound one. We're making every effort to give developers the best support possible, but compromises are sometimes necessary.

The decision not to supply the system software on the CD accompanying develop *was made independently of the MacWeek column and before it appeared. The timing was unfortunate, but it's not easy, or particularly practical, to reverse company decisions based on what MacWeek chooses to publish. (I hope you don't really think the column motivated us to remove the software from the CD!) We informed the MacWeek columnist immediately of what was about to happen, and he let his readers know. It did occur to us that developers subscribing in*

DO THE WRITE THING

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 303-4DP, 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).•

response to that column might be angry. But most of them turned out to be happy to receive develop *and all it does have on its CD for $30 a year. The majority of our subscribers have been understanding about the CD change, saying it always did seem like too good a deal to be true.*

I'm always sorry to hear from an unhappy camper. But thanks for writing.

—Caroline Rose

NEW CD PACKAGING: SO-SO

Your new packaging for the Bookmark CD sent with *develop* Issue 14 shows an appreciated effort for environmentally conscious packaging.

Unfortunately, this packaging is not U.S. Postal Service–friendly. I was able to flatten the CD enough so that it's working fine. Using a stiffer mailer and including the words "DO NOT BEND" clearly visible for the mail carrier would help prevent the CD from being damaged.

develop has been a great assistance to my business. I look forward to each and every issue.

Thank you for your assistance.

—Edward Salm

Sorry about your delivery problem. We've had a couple of other complaints about this, though not enough to justify the expense of implementing a solution like the one you suggest. We'll keep your words (and the unsuspecting mail carrier) in mind as we continue searching for packaging options that work without increasing the price of a develop *subscription.*

—Diane Wilcox

WHITHER THE DOGCOW?

I can't fight it anymore; I have to ask. In the Letters column in *develop* Issue 10, you say the story of the dogcow is hidden in Tech Note #31 on the CD. I found Tech Note #31; it was funny, but contains no references to "dog" or "cow" or "Moof!" What gives? I need to know the story of the dogcow!!!!

—Gary Robinson

The Tech Note you saw is actually #31A, "GestaltWaitNextEvent," which indeed is not on the subject of dogcattle. The original Tech Note #31, "The Dogcow," is no longer available, but I can give you some clues on where you might find a copy. It used to be hidden in the Technical Notes Stack on the early versions of develop*'s CD (most notably "Phil and Dave's Excellent CD"), although no one here seems to be exactly certain when it stopped. It appeared on paper only once, as part of the monthly mailing to Apple Associates and Partners back in April of 1989. Assuming you're not that far behind on reading your mail, you may want to try trolling the net. Macintosh programmers are an unusual breed, and I'm sure you'll find someone who has a copy.*

Why the continued secrecy? The answer has a little bit to do with history and a lot to do with tradition, and may or may not have to do with an exchange of spies during the Cold War. But there's good news: your letter has inspired me to write up how the dogcow furor and Tech Note originated — Apple cultural minutiae that may be of interest to other crazed Macintosh developers. The editor threatens to publish it in a future issue, assuming it passes by our censors.

—Mark ("The Red") Harlan
Author, Tech Note #31

SUBSCRIPTION INFORMATION

Subscriptions to *develop* are available through APDA (see inside back cover for APDA information), or you can 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 *develop Bookmark* CD, the *Developer CD Series* disc, and the Developer Services bulletin board on AppleLink.•

MAKING THE LEAP TO POWERPC

Apple will soon be introducing the first Macintosh CPU architecture not based on a 68000-family microprocessor. The entirely new architecture is built around a new RISC CPU — the PowerPC microprocessor jointly designed by IBM, Motorola, and Apple. Truly taking advantage of PowerPC technology will require an ongoing effort by both Apple and developers. Apple is making the first leap to this new platform; now it's up to developers to make the next leap and bring the performance made possible by PowerPC technology to their applications.

DAVE RADCLIFFE

In 1984, Apple Computer offered a startling vision of the future of personal computing by introducing the Macintosh, which radically changed the desktop. Now, nearly ten years later, the computing world embraces graphical interfaces. Ten years is a lifetime in computing terms; at that age, many computing architectures are considered ancient. The Macintosh enters its second decade by looking to the future while remembering its past — making the transition from the sturdy Motorola 68000 family to the sleek new PowerPC processor–based family without forsaking developers and users and their investment in the 680x0 architecture.

The PowerPC microprocessor is the most significant change to date in the Macintosh product line. This article introduces the new PowerPC architecture and discusses the ramifications for existing applications, as well as opportunities for new or revised applications to take full advantage of the power of the new chip. It contrasts the new architecture with the old and explains how this new architecture both acknowledges the past and prepares for the future.

COMPARING CISC AND RISC

Much has been written about the differences between a CISC (complex instruction set computer) architecture, used in Motorola's MC680x0 processors, and a RISC (reduced instruction set computer) architecture, used in the PowerPC microprocessor. The relative merits of the two architectures have also been widely

DAVE RADCLIFFE is a five-year veteran of Apple's Developer Technical Support group and for the past year has been excited to be part of the PowerPC project. When he's not plumbing the depths of MacsBug, Dave enjoys relaxing with a mug of beer from one of the local microbreweries, watching old movies at the Stanford Theatre, or just being a couch potato in front of one of his laser disc videos. For his sabbatical, Dave is looking forward to a low-tech adventure river rafting through the Grand Canyon.•

debated. A detailed discussion of CISC and RISC is beyond the scope of this article, but some understanding of RISC principles is useful for understanding PowerPC architecture.

Two logical considerations motivated CISC development. The first was a desire to simplify assembly-language programming by enriching the functionality of the instruction set. CISC architectures did this by providing a greater variety of instructions, as well as a wide array of addressing modes, thereby reducing the number of steps required to perform a particular operation. Second, as writing compilers became easier, there was a desire to provide instructions more closely related to operations performed by high-level languages. CISC architectures were marvelously successful at satisfying this goal also.

In the early 1980s, hardware designers began to run into the limitations inherent in CISC architectures, particularly in their ability to streamline the flow of instructions. At the same time, the software world was deemphasizing assembly-language programming in favor of high-level languages with sophisticated, optimizing compilers. This allowed hardware designers to simplify their architecture and shift much of the performance burden to compiler writers.

The classic equation for execution time is

$$ET = \sum_{i=1}^{N} CPI * CT$$

where *ET* is the total execution time, *N* is the number of instructions executed, *CPI* is the number of cycles per instruction, and *CT* is the cycle time. Both CISC and RISC architectures benefit from reduced cycle time. Faster clock rates translate directly to smaller cycle times, and hence shorter execution times. Where CISC and RISC architectures differ is in their approach to *N* and *CPI*. CISC tries to shorten execution times by minimizing *N*, while RISC tries to minimize *CPI*.

PIPELINING

The four typical stages in executing an instruction are fetch, decode, execute, and write. In a simplistic architecture, these stages all happen in sequence, and the next instruction can't start until the previous instruction has finished, as shown in Figure 1. Designers realized that this need not be the case and that each of these stages can overlap. Once an instruction is fetched and passed to the decode stage, the next instruction can be fetched without waiting for the first instruction to complete. This technique, known as *pipelining*, is shown in Figure 2.

The example in Figure 2 executes the same two instructions, but in only nine cycles, compared to 12 cycles in the nonpipelined case. There's a curious thing about this example, though: the second instruction takes eight cycles to complete when pipelined, but only five when it's not. This is because the various stages take different

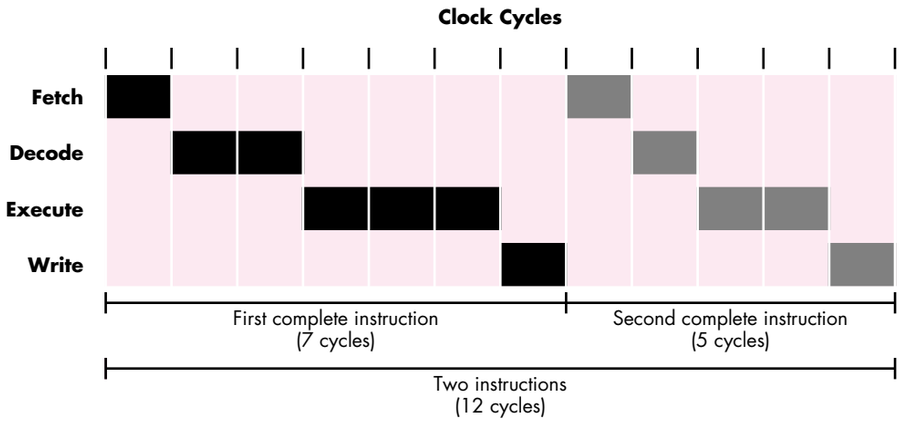


Figure 1
Nonpipelined Stages of Execution

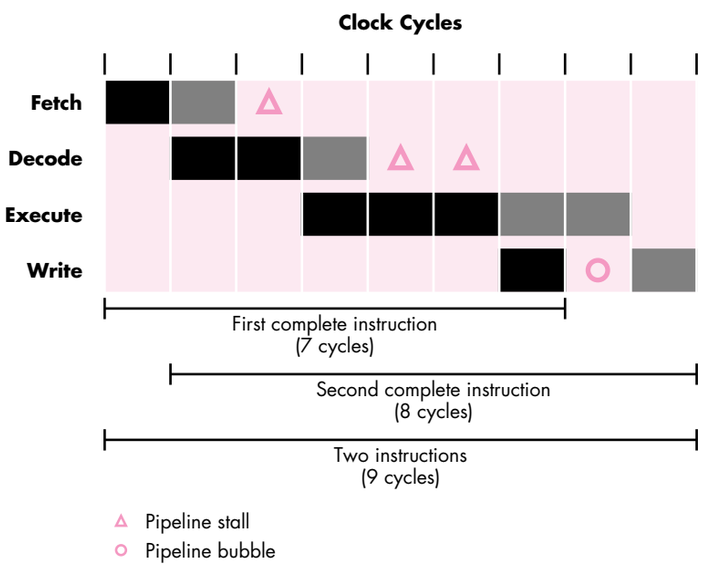


Figure 2
Pipelined Stages of Execution

amounts of time to complete. The overall result is better, but unnecessary delays can occur in instruction execution.

Variable numbers of cycles per stage is a characteristic of CISC architectures. Complex instructions may occupy multiple words, requiring multiple cycles to fetch. Multiple operands complicate the process of decoding. More complicated instructions take longer to execute than simpler instructions. In Figure 2, the execute stage of the second instruction is delayed two cycles while waiting for the first instruction to execute. This is known as a pipeline *stall*. Similarly, the write stage sits idle for one cycle between the first and second instructions while waiting for the execute stage of the second instruction to complete. This is known as a pipeline *bubble*. Both stalls and bubbles reduce the efficiency of the pipeline and increase the overall number of cycles per instruction.

INCREASING PIPELINE EFFICIENCY

RISC architectures work very hard to eliminate inefficiencies in the instruction pipeline and keep the pipeline jammed full. RISC architectures share most or all of the following common features:

- Instructions are a uniform length. Variable-length instructions in CISC architectures mean that time must be spent just figuring out how long the instruction is and how many operands it uses. RISC architectures don't have that problem.
- Simplified instructions, instruction formats, and addressing modes allow for fast instruction decoding and execution.
- Relatively large numbers of registers and large amounts of fast-cache memory reduce cycles spent for access to slower, main memory and allow frequently used variables to be kept loaded.
- Load/store architecture is used for access to memory. The only memory-to-register and register-to-memory operations are load and store instructions. All other operations are register only. Register-to-memory and memory-to-memory operations in CISC architectures require multiple cycles to complete.
- Instructions are simple. In an ideal RISC machine, each stage requires one cycle to complete.
- For improved performance, instructions can be implemented directly in hardware instead of being microprogrammed as in CISC processors.

Figure 3 shows an example of executing instructions on a nonpipelined RISC machine. When instructions are not pipelined, they complete serially, with two instructions completing in eight cycles. The optimal case for pipelining instructions is shown in Figure 4. Now you have the two instructions executing in just five cycles. If

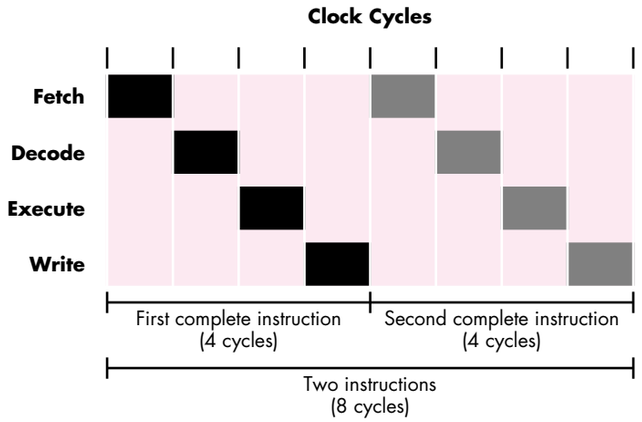


Figure 3
RISC Nonpipelined Stages of Execution

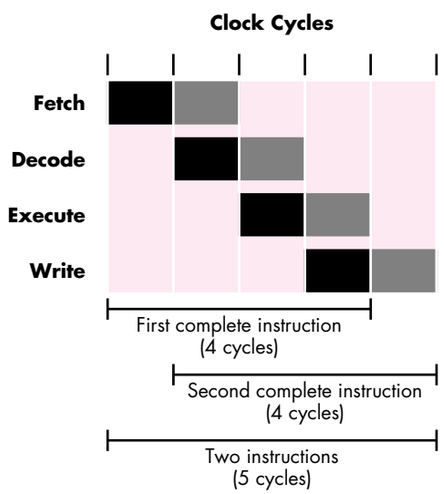


Figure 4
RISC Pipelined Stages of Execution

the pipeline is kept full like this, the number of cycles per instruction drops to just one. This is the goal of most RISC architectures.

One cycle per instruction is the ideal case for this example, but in reality, stalls and bubbles occur, even in the best architectures. This is where the compiler comes into play. The compiler has detailed knowledge of how the program should work. It need not perform operations in the order specified in the source code; it need only guarantee that the right result is obtained. If you build into the compiler some knowledge of how to make best use of the CPU, the compiler can make a huge difference in program performance.

Consider the following two C instructions:

```
b = *a + 5;
d = *c + 10;
```

The variables *a*, *b*, *c*, and *d* are all long or pointer-to-long variables. The compiler might generate the following assembly instructions on the PowerPC microprocessor:

```
lwz   r5,0(r3)       ; Load value pointed to by r3 into r5
addi  r5,r5,0x0005   ; Add 5 to value in r5
lwz   r6,0(r4)       ; Load value pointed to by r4 into r6
addi  r6,r6,0x000a   ; Add 10 to value in r6
```

The **lwz** instruction (Load Word and Zero) loads a register from a source value. On a PowerPC processor, words are 32-bit values; 16-bit values are half words. The **addi** instruction (Add Immediate) adds the immediate value and stores the result.

Figure 5 shows what happens when these instructions execute. Both **addi** instructions stall in the decode stage because they can't enter the execute stage until the register is available from the **lwz** instruction.

The compiler can prevent the stalls. Instead of following the flow of the original source code, you can rearrange the instructions as follows:

```
lwz   r5,0(r3)       ; Load value pointed to by r3 into r5
lwz   r6,0(r4)       ; Load value pointed to by r4 into r6
addi  r5,r5,0x0005   ; Add 5 to value in r5
addi  r6,r6,0x000a   ; Add 10 to value in r6
```

Now look at what happens to the instruction pipeline (Figure 6): there are no delays. By moving the add instructions to later in the instruction stream, you allow the load instructions they depend on to complete, so the add instructions can execute immediately.

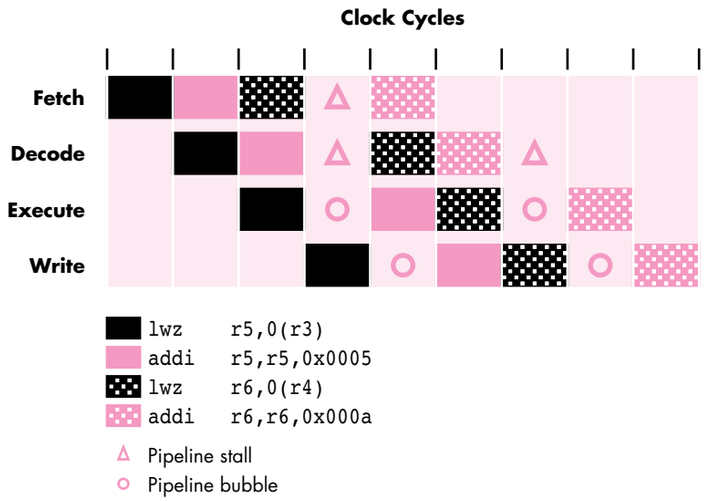


Figure 5
Stalled Pipelined Execution

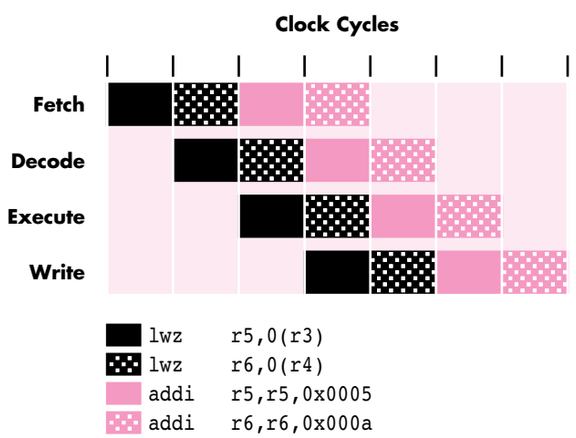


Figure 6
No-Delay Pipelined Execution

BRANCHING

All pipelined architectures face the problem of branches. Any time a conditional branch is encountered, the processor faces a dilemma because now two instruction streams are possible. It can't pipeline both possible paths. It can guess which path to take, but if it guesses wrong, the pipeline is disrupted.

One common approach to this problem is a technique called *delayed branching*. In delayed branching, the processor *always* executes the instruction immediately following the branch instruction. While starting this instruction, the CPU can be figuring out the destination of the branch instruction and so can keep the pipeline flowing. Of course, it's important that the instruction after the branch not affect the branch. It's up to the compiler to find an instruction unrelated to the branch instruction to fill this delay slot. If it can't fill the delay slot, the compiler can always put in a no-op instruction, but this is inefficient. Some architectures allow the instruction in the delay slot to be ignored if the branch is taken. This avoids the need to fill the delay slot with a no-op instruction, but undermines the purpose of delayed branching. PowerPC architecture takes a unique approach to the branching problem, as discussed later in the section "Branch Processor."

SUPERSCALAR DESIGN

Another technique RISC designers use to increase performance is superscalar or multi-issue design. The simpler design of RISC architectures makes it possible to build in multiple processing units; this is superscalar design. In the same way that the compiler can juggle instructions to avoid resource constraints, the CPU can now reduce bottlenecks and achieve higher performance by feeding instructions to separate processing units operating in parallel. This allows average instruction cycle times to drop below one cycle per instruction. PowerPC microprocessors use this technique as discussed later in the section "Functional Units of the PowerPC Microprocessor."

RISC ADVANTAGES

One last point needs to be made before leaving a comparison of CISC and RISC. Many of the techniques used by RISC designers can and are used by CISC designers. Modern CISC chips such as the MC68040 and Intel 80486 make extensive use of instruction pipelining, parallel integer and floating-point units, fast cache architectures, and resource constraint reduction (such as delayed writes) to achieve the performance they do. But the sheer complexity of the designs means they're hard to implement (and implement correctly), which often results in long development cycles. The simplicity of RISC architecture helps avoid this problem.

Similarly, the compiler can aid CISC machine performance. But the complexity of CISC design means it's nearly impossible to determine instruction timing, so it's difficult for the compiler to choose the best instruction sequence. Instruction scheduling is also possible but more difficult. The finer granularity of the RISC

instruction set gives the compiler much more flexibility and control over the resources provided by the CPU.

Simplified hardware and the influence of the compiler are really the ultimate advantages of RISC.

POWERPC CPU ARCHITECTURE

PowerPC architecture is a modern 64-bit, RISC architecture adhering to all the previously discussed design goals. It has 32 general-purpose and 32 floating-point registers. All instructions have a uniform 32-bit length. The first PowerPC microprocessor, the PowerPC 601, is a superscalar implementation of the 32-bit subset of this architecture.

POWERPC VERSUS POWER

The PowerPC microprocessor is a single-chip design descended from an earlier, multichip IBM RISC implementation known as POWER. It's worth mentioning the differences between the two architectures.

- Misaligned data access. Most RISC architectures require all data access to be word (4-byte) aligned. POWER was ambiguous regarding data alignment. PowerPC architecture explicitly allows misaligned data access but with a possible performance penalty. The advantage is that it allows use of data structures aligned for 680x0 architecture.
- Elimination of the MQ register. POWER has a special-purpose multiply/quotient (MQ) register for extended-precision integer arithmetic. But since there's only one register, it becomes a bottleneck that hinders superscalar implementations. The MQ register, and all instructions that depend on it, were eliminated from the PowerPC architecture.
- Addition of single-precision floating point. POWER supports only double-precision floating point. PowerPC architecture supports single precision as well, which may be more appropriate for some applications. (There's no hardware support for 80- or 96-bit extended floating point, which 680x0 developers are familiar with. The consequences of this for developers are discussed in "Native PowerPC Numerics.")
- 64-bit architecture. POWER is a 32-bit architecture. PowerPC architecture is fully 64 bit; however, the first implementations feature a 32-bit subset of the architecture. Code written for 32-bit processors will be fully supported on 64-bit implementations running in 32-bit mode.

NATIVE POWERPC NUMERICS

BY ALI SAZEGARI

Developers dependent on floating point who port to the PowerPC platform will enjoy superior floating-point performance. However, some special consideration is needed, because the floating-point implementation on the PowerPC processor differs from that of the 680x0 processors.

POWERPC ARCHITECTURE FEATURES

The PowerPC microprocessor floating point is an IEEE 754–compliant single- and double-precision implementation offering fast, pipelined, nondestructive floating-point operations. These operations are add, subtract, multiply, divide, compare, convert to int, and a new class of multiply-add fused (MAF) instructions of the form

frT ← (*frA* * *frB*) + *frC*

where *fr* is a floating-point register. In MAF operations, all bits of the resultant multiply section are kept (106 bits in double) and participate in the final rounding, producing a more exact result. In other words, (*A* * *B*) + *C* is a single operation with one rounding. The compilers on the PowerPC platform use MAF instructions wherever possible, unless expressly prohibited by the user.

The PowerPC microprocessor has a rich set of floating-point register files: 32 floating-point double-precision data registers and a combined status and control register (unlike the MC6888x or MC68040).

C PROGRAMMER'S MODEL

The PowerPC microprocessor shared math library, MathLib, complies with the emerging Floating-Point C Extensions (FPCE X3J11.1/93-001) of the Numerical C Extensions Group (NCEG) specification. FPCE extends C to provide access to floating-point features generally and IEEE 754/854 specifically. FPCE provides a superset of math.h and sane.h functionality. The new required include files are fp.h and fenv.h.

The FPCE fp.h file is a collection of mathematical functions. It defines all math.h and nonenvironmental sane.h functionality plus hyperbolic, inverse hyperbolic, max, min, positive difference, error, and gamma functions. Other functions round floating-point numbers to integral values or integral format. An extensive array of correctly rounded binary-to-decimal conversion functions is provided.

The FPCE fenv.h file defines all the functions used to query or modify the floating-point environment (exception flags and rounding direction).

The include file math.h is kept for ANSI C compliance, but developers are encouraged to use fp.h and fenv.h. The sane.h include file won't be supported. *Be aware of function name and prototype differences between SANE and FPCE-NCEG interfaces*. For example, the functions copysign and scalb have reversed arguments in the new fp.h, and log1 is now called log1p.

FP DATA TYPES

Table 1 lists the available *native* data types on the PowerPC microprocessor. There's no hardware or compiler support for the 80- or 96-bit IEEE extended values commonly used by Macintosh programmers. Developers should use 64-bit double as their native data type and use rescaling techniques within their algorithms susceptible to numerical ill-conditioning. The 64-bit comp type, a floating-point data type available on the 680x0-based Macintosh, isn't supported. Use the data type long double judiciously and only when an algorithm requires the extra precision. SANE data types, which include extended and comp, are fully supported in emulation mode on PowerPC processor–based Macintosh systems.

The transcendental long-double functions are not supported for the first release of MathLib on PowerPC processor–based Macintosh systems. A complete long-double library is planned for a later release.

Thanks to Paul Finlayson and Stuart McDonald for their review of "Native PowerPC Numerics."•

Table 1
Available Native Data Types on the PowerPC Microprocessor

Native Data Type	Description
float	IEEE single precision (32 bits with fast operations)
double	IEEE double precision (64 bits with fast operations)
long double	128-bit structure of two doubles (head and tail), whose value is head + tail. Not an IEEE double-extended type! Provides additional precision within double range.

Note: The long double data type isn't supported by the hardware, so operations are relatively slow. It should be used selectively.

PowerPC architecture uses big-endian byte order, just like 680x0 and POWER. As an added feature, it also supports a mode using little-endian byte ordering and provides instructions to allow access to little-endian data from big-endian mode and to big-endian data from little-endian mode.

FUNCTIONAL UNITS OF THE POWERPC MICROPROCESSOR

Figure 7 is a block diagram of the PowerPC 601 microprocessor, the first member of the PowerPC processor family. This microprocessor is a superscalar PowerPC implementation, with three separate execution units: the fixed-point and floating-point units and the branch processor. The branch processor initiates instruction execution by fetching instructions from the instruction cache (which is filled from memory if there are no instructions in it). The branch processor then feeds integer and floating-point instructions to the fixed-point and floating-point units respectively. These units operate on data in registers and in the data cache (which is filled from memory if there's no data in it). The fixed-point unit is also involved in address decode operations.

BRANCH PROCESSOR

The branch processor deserves special attention. As mentioned earlier, PowerPC architecture takes an original approach to the problem of branch penalties, and the branch processor is responsible for this. The branch processor contains within it everything needed to determine how to handle a branch instruction. This includes three special-purpose registers:

- The condition register (CR) has flags set by certain operations and is used for conditional branching.
- The link register (LR) can contain a destination address for a branch instruction and can also hold the return address after branch and link (subroutine) instructions.
- The count register (CTR) is used for looping and indirect branches.

For divisible integer quantities composed of separately addressable bytes — for example, a 32-bit integer subdivided into four addressable bytes — there are numerous ways to arrange the bytes. Only two arrangements make sense and are in use on computers today. Big-endian byte ordering means the most significant byte (the big end of the number) is assigned the lowest address. Little-endian byte ordering means the least significant byte is assigned the lowest address; it's used, for example, on Intel 80x86 CPUs. The terms originated in Jonathan Swift's *Gulliver's Travels*, where the controversy was over breaking an egg at the big end or the small end.•

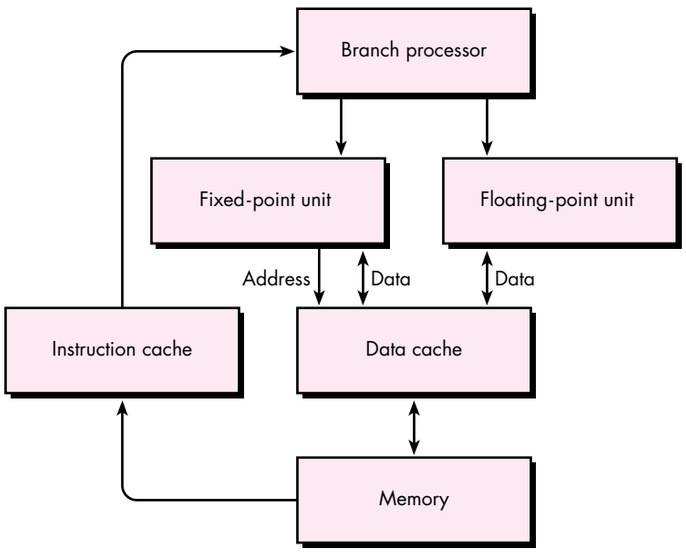


Figure 7
Block Diagram of PowerPC 601 Chip

For unconditional branches, the branch processor knows unambiguously which path to take. For conditional branches, if a branch condition is set far enough before the actual branch instruction, the branch processor has the information necessary to determine which path to take.

The design of the condition register uniquely aids the processing of conditional branches. Instead of a single set of condition codes, it contains eight 4-bit condition code fields, designated CR0, CR1 . . . CR7. Compare operations allow each field to be set independently. A compiler using these multiple, independent condition code fields has more flexibility in scheduling instructions to assist the branch processor. As an additional performance enhancement, instructions that might set condition codes (such as **add**) do so only if a record bit is set in the instruction, so time isn't spent setting condition codes that would otherwise be ignored.

The branch processor also has knowledge of the count register, used in looping operations. This lets the branch processor know in advance when a loop will finish.

With this design the branch processor can preprocess the instruction stream and, in most cases, determine in advance the target of the branch operation. This allows it to "fold" the branch instruction out of the instruction stream, so the fixed-point and

floating-point units see an unbroken stream of instructions and fewer branch penalties occur.

POWERPC RUNTIME ARCHITECTURE

An important goal in the development of Apple's PowerPC processor–based machines was to preserve user and developer investment in the 680x0 architecture. Another important goal was to port the existing 680x0 Toolbox and operating system to the new platform quickly. Both goals were met through the ability to emulate 680x0 instructions in software on the PowerPC microprocessor. So the first way to view a Macintosh on PowerPC, and indeed the way existing applications and system software view this machine, is as a 680x0-based Macintosh. In this section we approach this new beast through the 680x0 emulator and then peel away the layers to reveal the underlying PowerPC runtime architecture.

SOFTWARE EMULATOR

The software emulator understands and executes the instruction set of a Motorola MC68020 processor. You might wonder why Apple chose to emulate the MC68020 and not the latest and greatest processors such as MC68030 and MC68040.

- The only advantage of the 68030 over the 68020, in terms of instruction set, is the integrated memory management unit (MMU). The MMU is really for use by the operating system for implementing features such as virtual memory. The PowerPC microprocessor MMU operation is very different from 680x0 MMU operation, and there's no need for applications to execute MMU instructions anyway. Applications needing control over virtual memory can still use the existing virtual memory interface; just the implementation will be different.
- Similarly, the key advantage of the MC68040 over its predecessors is the integrated floating-point unit. The PowerPC microprocessor has its own floating-point implementation. Apple already provides a standard numeric interface for 680x0 applications, called SANE, and emulating floating-point instructions using native PowerPC code offers no real advantages over implementing SANE as native PowerPC code.

As a bonus feature, the emulator also supports certain advanced user-mode instructions such as the MOVE16 instruction from the MC68040. However, from a programmer's point of view, the emulator behaves as an MC68020 (for example, Gestalt reports an MC68020 is present) and developers are advised not to take advantage of any features outside the MC68020 architecture.

Once the emulator was up and working, the PowerPC processor–based machine almost immediately gained an operating system, since all the code in the ROM and

the operating system was now executable. This also gave the machine a high degree of compatibility with older Macintosh models, because the same code, with all its idiosyncrasies, is being executed.

Had Apple stopped here, you'd have a machine that works great but is pretty boring. After all, who wants a machine that pretends to execute 680x0 code, but not necessarily as fast as the real thing? Why not get a real 680x0 machine instead? The answer, of course, lies in tapping into the power behind the emulator — the PowerPC microprocessor itself.

TOOLBOX ACCELERATION

All Macintosh applications spend part of their time calling the Macintosh Toolbox. In turn, the Toolbox performs the requested service by executing Toolbox code on behalf of the application. You can think of the Toolbox as an extension of the application. The advantage of this during development of PowerPC processor–based machines is that selectively replacing portions of the Toolbox with equivalent PowerPC code greatly enhances the performance of those portions of the Toolbox. All applications that use those routines benefit from improved performance. No modification of the application is required to receive the benefit.

Ideally, of course, it would be best if the entire Toolbox executed as native code. But that requires a huge amount of work and would delay the first release of Macintosh on PowerPC. Analysis of application programs revealed that some portions of the Toolbox are used more heavily than others. All applications, for example, rely heavily on QuickDraw. Effort spent porting QuickDraw would benefit more applications than, say, porting the Dialog Manager. So the first release of Macintosh on PowerPC will target the portions of the Toolbox that will provide the greatest performance enhancement to the greatest number of applications.

As Apple releases new versions of the system, with more and more of the Toolbox as native PowerPC code, users will magically get a "faster" machine without adding new hardware. All they have to do is install the newer, accelerated Toolbox.

At the same time, the goal is not just to enhance the performance of the system, but to empower application software as well. The accelerated Toolbox is a start, but real PowerPC application performance comes from having native PowerPC applications, and the first release of Macintosh on PowerPC will include an entirely new runtime architecture in support of native applications.

WHY A NEW RUNTIME ARCHITECTURE?

The new runtime architecture addresses many of the following limitations of the 680x0 architecture:

- The first Macintosh models were severely limited in the memory available to applications, so the runtime architecture was designed

to squeeze the most out of the memory that was available. Today, the relative availability of cheap RAM removes this limitation.

- Hard disks and memory management units required to support virtual memory were unavailable, so applications were required to load discrete blocks of code through the Segment Loader. With the relative availability of cheap RAM and support for virtual memory, most reasons for having the Segment Loader disappear.
- The system now supports a wide variety of code types — not just applications and system software, but standalone code blocks, such as INITs and MDEFs, and loadable code plug-ins, such as XCMDs and components. These code blocks strain the runtime architecture because it's difficult to manage global data for these blocks and to import and export functions between blocks.
- There's a large amount of code duplication in the Macintosh. The Toolbox provides some code sharing between applications, but in general, most applications have built into them large amounts of redundant code. For example, library and glue code gets linked into every application. Having it built into the application increases demands on disk and memory resources because each instance of the application must have the duplicated code.

CODE FRAGMENT MANAGER

The centerpiece of the new architecture is the Code Fragment Manager. Each block of executable PowerPC code is a code fragment. A code fragment is autonomous, with its own static data. It can export both code and data references for use by other fragments and import code and data references from other fragments for its own use. Because such references are resolved at run time, code fragments are a form of dynamically linked, shared libraries. (See "Code Fragment Manager or Shared Library Manager?" for an explanation of the relationship between the two managers.)

From a native PowerPC application's point of view, access to the Macintosh Toolbox now occurs through a shared library maintained by the Code Fragment Manager. Applications no longer have segments — they have one or more code fragments. The main code fragment is loaded at launch time and any external references to other shared libraries are resolved. An application neither knows nor cares whether a reference is internal or external; access is completely transparent.

In some cases applications may want to manage code fragments on their own. For example, standalone code resources can now be handled as code fragments. This makes code resources such as XCMDs much easier to develop. Not only does such a resource have its own static data, but function references within the resource are fully exportable. Complicated parameter blocks aren't needed for passing data or jumping into the beginning of a code resource. Furthermore, because the application code is

CODE FRAGMENT MANAGER OR SHARED LIBRARY MANAGER?

You may already be familiar with an implementation of shared libraries for the Macintosh known as the Shared Library Manager. The advantage of the Shared Library Manager is that it works with today's 680x0 runtime architecture. The Code Fragment Manager, on the other hand, lays the foundation for a new and more modern runtime architecture.

The first releases of these two managers will be mutually exclusive. The Shared Library Manager will be implemented only for 680x0 and the Code Fragment Manager will work only on the PowerPC microprocessor.

In the future, though, the Code Fragment Manager will be available on 680x0-based machines as well, and a future release of the Shared Library Manager (version 2.0) will be built on top of the Code Fragment Manager. This will provide Shared Library Manager support for Macintosh on PowerPC. Developers should code for whichever mechanism best suits their needs and target platform.

itself a code fragment and can export its references, the standalone code has access to functions and data within the application itself. Complicated callback mechanisms are no longer necessary.

MIXED MODE MANAGER

There's one final piece to the PowerPC architecture puzzle. The Macintosh Toolbox makes wide use of pointers to functions. FilterProcs, I/O completion routines, A-trap vectors, QuickDraw bottlenecks, definition procedures (such as MDEFs, MBDFs, and CDEFs), and other types of standalone code (such as INITs and VBL tasks) are just a few examples of the wide variety of function pointers in use on the Macintosh.

On a 680x0-based Macintosh, life is easy because a function pointer is just the address of a 680x0 routine that can be called. On a PowerPC processor–based Macintosh, life is much more complicated; not only is the Toolbox a mixture of 680x0 and PowerPC code, but a function pointer could be a pointer to 680x0 code or PowerPC code and the caller should neither know nor care what kind of code it's calling.

To handle this situation, Apple is introducing the Mixed Mode Manager. One problem that this manager must solve is the mismatch between calling conventions for 680x0 and PowerPC code. PowerPC code follows C conventions, with parameters passed right to left. The 680x0 code uses a variety of calling conventions: some traps are register based while some are Pascal stack based with parameters passed left to right. The Mixed Mode Manager must make calls between disparate functions seamless. Furthermore, it must do it in a way that's compatible with existing 680x0 applications. Since existing binaries must work unmodified, the existence of the Mixed Mode Manager must be completely transparent to these applications.

The Mixed Mode Manager's task is shown in Figure 8. Instead of passing a function pointer of type ProcPtr to the Toolbox, applications must now pass a function pointer

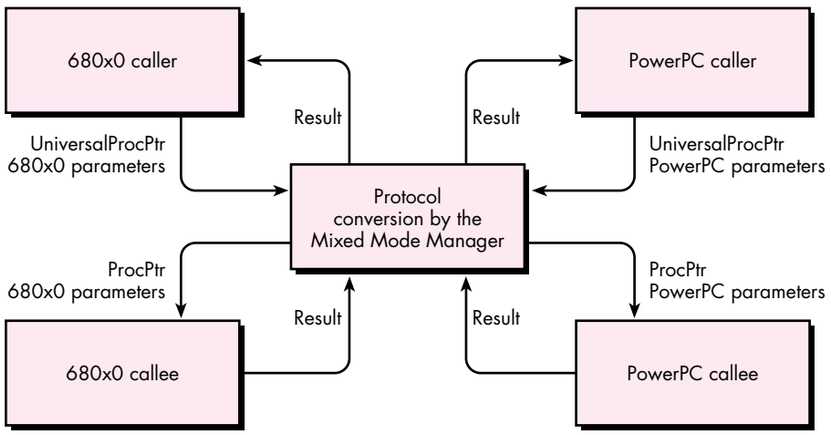


Figure 8
Mixed Mode Manager

of type UniversalProcPtr. UniversalProcPtr is a generic version of ProcPtr that lets the Mixed Mode Manager know how to route the call. Whenever 680x0 or PowerPC code calls a function through a UniversalProcPtr, the Mixed Mode Manager looks at the destination for the call. If a mode switch isn't necessary — in other words, if both the caller and the callee are the same code type — the Mixed Mode Manager does nothing and just passes the call to the caller.

If a mode switch is necessary — in other words, if a 680x0 caller is calling PowerPC code, or vice versa — the Mixed Mode Manager performs a protocol conversion, rearranging the parameters, including moving them into or out of registers as necessary to ensure that the callee sees the parameters correctly. When the callee returns, the Mixed Mode Manager performs a protocol conversion in the other direction to ensure that return values are correctly passed back to the caller.

For 680x0 applications, the Mixed Mode Manager is completely transparent and these applications run without modification. PowerPC applications, however, must become aware of the Mixed Mode Manager. The basics of using the Mixed Mode Manager are covered along with UniversalProcPtrs later in the section "UniversalProcPtrs."

WRITING PORTABLE C CODE

The preferred development languages for PowerPC code are C and C++. Therefore, the first step in preparing for the PowerPC platform is to provide portable C and C++ code. The examples here use C, but the principles apply to C++ as well.

The compilers for PowerPC C code are stricter than either the MPW or the THINK C compiler, so the best way to prepare your code for the PowerPC platform is to be sure it follows the ANSI C standard. You should take full advantage of the stronger type checking and prototyping features an ANSI C compiler provides.

Consistent use of function prototypes is the best way to ensure portable code. ANSI C prototypes fully qualify the parameters to a function, as shown in this example:

```
void DoEvent (EventRecord *event);
```

It's usually permissible to mix the new-style function declaration with the old-style function definition:

```
void DoEvent (event)
EventRecord *event;
{
   . . .
}
```

However, mixing function declarations in this way typically defeats the purpose of having a function prototype in the first place. So the first step in writing portable code is to be sure you consistently use ANSI C function prototypes throughout.

INTEGERS AND BITFIELDS

Variations in the size of integers of type int always cause trouble when you're trying to port code. This is more of a problem for THINK C code, which allows 16-bit integers of type int. C purists may not agree, but my recommendation is never to use type int. Always use integers of types short and long (or an equivalent type). The Macintosh Toolbox itself is explicit about data sizes, and experience has shown that developers dependent on the THINK C 16-bit integers of type int have more difficulty porting to the PowerPC platform.

A similar caution applies to bitfields. Bitfields are useful for access to machine-dependent data structures and the like, but are inherently implementation defined and therefore nonportable.

DATA STRUCTURES

Some compilers allow incomplete arrays as the last member in a data structure:

```
struct QElem {
   struct QElem   *qLink;
   short          qType;
   short          qData[];
};
```

This isn't allowed by the ANSI C standard. Here's a more portable definition:

```
struct QElem {
   struct QElem   *qLink;
   short          qType;
   short          qData[1];
};
```

Similarly, some compilers allow comparison of data structures. Again, this isn't allowed by the ANSI C standard, so attempting to do something as simple as comparing two Rects will fail on the compilers for PowerPC code.

When using data structures, you need to be aware of data alignment. RISC machines prefer (and often require) that data be aligned on a 4-byte boundary. But on the 680x0, the default is to align data to a 2-byte boundary. PowerPC architecture specifically allows misaligned data access, but there can be a small performance penalty if multiple bus cycles are required for access to the data. This creates a dilemma: portability versus performance.

Because the Macintosh Toolbox relies on 680x0 data structures, data passed to the Toolbox must have 680x0 alignment. The same applies if you want to share data with 680x0 applications. To solve this, the compiler now allows you, through #pragma statements and compiler options, to align PowerPC code data structures just like 680x0 code data structures. But if the structure is only internal to your application, you probably want to use the natural PowerPC code alignment. Although it's likely to be painful to modify existing data structures for PowerPC code alignment, if you're designing new data structures, you can keep the alignment issue in mind and create structures that are optimal for both 680x0 and PowerPC processor–based machines.

COMPILER EXTENSIONS

In addition to supporting 680x0 data alignment, compilers for PowerPC code have been extended in several other ways to make porting easier. This involves supporting several of the MPW C compiler extensions and features:

- The compiler understands "\p" at the start of a string for the generation of Pascal strings.
- The **pascal** function keyword is allowed by the compiler, but ignored. A subtle consequence of this is discussed in the section "Pascal Functions."
- The compiler won't complain if you use C++ style line-end comments (//).
- MPW C packs enums into the smallest data type possible and the compilers for PowerPC code have been extended to support the feature.

How can you tell if your code is ANSI C compliant? You can eliminate many of the idiosyncrasies in your code by compiling it with multiple compilers. Code conditioned in this way is much more portable to the PowerPC platform than code dependent on a single compiler. So one of the best ways to prepare for the PowerPC platform is to make sure your code compiles and runs with both MPW C and THINK C.

WRITING CODE FOR POWERPC

Some changes to the programming model are necessary for the development of PowerPC code. However, Apple tried to limit changes so as to make the transition to the PowerPC platform easier for developers (see "Universal Interfaces" to understand how these changes affect development for 680x0 platforms).

COMPATIBILITY GUIDELINES

Everything ever written about compatibility guidelines for the Macintosh applies to the Macintosh on PowerPC in spades. Here are some of the key points:

- The code must be 32-bit clean. Most applications now satisfy this requirement, thanks to System 7, but it deserves reiterating because 24-bit mode will no longer be an option.
- For the first release of Macintosh on PowerPC, access to low memory is allowed exactly as before. Direct access to low memory applies for both 680x0 and native PowerPC applications; however, a procedural interface is provided as part of the new API, and developers are strongly urged to begin using it for future compatibility. For example, CurDirStore is a commonly used low-memory global, and two new functions are defined to provide access to it:

```
long LMGetCurDirStore (void);
void LMSetCurDirStore (long CurDirStoreValue);
```

- Don't depend on undocumented data structures. Also, don't depend on alignment of data structures.
- Don't write data into code. In the past, this was often necessary because of limitations of the runtime architecture. Many of the reasons for doing it no longer exist with PowerPC architecture, so avoid it.
- Beware of dependencies on floating-point data types (see "Native PowerPC Numerics," earlier in this article).
- Don't depend on the hardware. Not only is there no longer a 680x0 CPU present, but the I/O architecture can also change. Use programmatic interfaces to perform I/O.

UNIVERSAL INTERFACES
BY DEAN YU

As Apple takes the Macintosh experience to a new chip architecture, it becomes more important than ever to have portable source code. With that in mind, Apple has created a set of *universal interface files*, which are provided on this issue's CD. The same interface file — for example, Windows.h — can be used to compile any source file for a Macintosh on either a 680x0 or a PowerPC microprocessor. The main changes you'll find in the C universal interface files are described below.

All system software routines declared extern. On the PowerPC platform, all routines can potentially be in a shared library, so all routines must be declared extern in order for the compiler to generate the correct code. Declaring routines extern is also compatible with MPW C.

Inline code wrapped in macro definitions. Obviously, 680x0 inline code isn't very useful on a PowerPC platform. 680x0 inline code is isolated by macros such as THREEWORDINLINE, which are defined in ConditionalMacros.h. These macros expand to inline initializers when compiling for 680x0 on non–shared library based platforms, and do nothing when compiling for PowerPC or shared library–based platforms.

UniversalProcPtrs. As discussed more fully in this article, the biggest change in the interface files is the introduction of the UniversalProcPtr data type used by the Mixed Mode Manager. In support of cross-platform code generation, the interface files define special "New" and "Call" macros (such as NewGrowZoneProc and CallGrowZoneProc) that hide the implementation details of using UniversalProcPtrs. For example, when you compile your application as 680x0 code, the Call macros jump to the routine pointed to by the UniversalProcPtr directly rather than invoke CallUniversalProc as they would for PowerPC compilation. Note that 680x0 versions of the Call macros are provided only for stack-based ProcPtrs.

Low memory access. To isolate dependencies on low memory, the SysEqu.h file has been removed and replaced by LowMem.h, which defines accessor functions for low memory. Previously defined accessor functions, such as MemError, are still defined but call through to the new accessor functions when appropriate.

Structure alignment. To maintain data structure compatibility, structs follow 680x0 word alignments when being compiled for the PowerPC microprocessor.

Even if you don't plan on porting your application immediately to the PowerPC platform, you can begin using the universal interface files for 680x0 development and make a crucial step toward future PowerPC compatibility.

- Don't depend on the 680x0 runtime model, which is very idiosyncratic. Fortunately, many of those idiosyncrasies were eliminated in the PowerPC runtime architecture, making your life easier but complicating the move from the 680x0 to this new architecture.

Some of these points are discussed in the following sections.

REVISITING THE CODE FRAGMENT MANAGER

As previously mentioned, the centerpiece of the PowerPC runtime architecture is the Code Fragment Manager. Rather than having a collection of code resources, a

PowerPC application has a code fragment (generally one, but possibly more) that lives in the data fork of the application. When an application is launched, the Process Manager determines whether a native PowerPC code fragment is present by looking for a 'cfrg' resource. This resource provides the necessary information for the Code Fragment Manager to load the main code fragment and resolve any external code and data references. The Code Fragment Manager also sets up global data for the code fragment.

The Code Fragment Manager eliminates the need for a segment loader. If virtual memory isn't present, the Code Fragment Manager loads the entire code fragment into memory; otherwise, it relies on virtual memory to page code directly in from the application when needed.

A 680x0 application maintains a notion of an A5 world, an integral part of the 680x0 runtime environment. Register A5 provides access to four kinds of data:

- application global data
- application QuickDraw global variables
- application jump table
- application parameters

Of these, only the QuickDraw global variables remain relevant. A wide variety of system and application code depends on using A5 to locate QuickDraw globals. Even though a native application has no use for a 680x0 register A5, the system still maintains an A5 world so that code that does depend on A5 has access to the right data. This means SetCurrentA5 and SetA5 will do the right thing with QuickDraw globals if you need to swap A5 worlds.

The 680x0 Macintosh Toolbox uses a wide variety of calling conventions. The two most common ones are Pascal stack based and register based. Variations include passing a selector to dispatch to a variety of functions or passing a pointer to a parameter block in register A0 (for VBL tasks, notification tasks, and I/O completion routines) or register A1 (for Time Manager tasks). Two of my personal favorites are the TextEdit highHook and caretHook routines: when called they have a pointer to the edit record in A3 and, instead of a return address, a pointer to a rectangle on top of the stack. The point is that it's nearly impossible to write 680x0 Macintosh applications entirely in a high-level language. Some assembly-language programming is required just to move these weird parameters around.

Life gets much easier on the PowerPC platform, which relies on uniform C calling conventions for everything. In almost all cases, 680x0 inline assembly and assembly wrapper routines can be rewritten in C for PowerPC code. For example, a 680x0 application can use the following assembly highHook routine to underline a selection:

```
HighHookUnderline
   MOVE.L   (SP),A0           ; Get the address of the rectangle
   MOVE  bottom(A0),top(A0)   ; Make the top coordinate equal to
   SUBQ  #1,top(A0)           ;  the bottom coordinate minus 1
   _InverRect                 ; Invert the resulting rectangle
   RTS
```

It's impossible to write this routine in C because of the weird calling conventions that supply the pointer to the Rect on top of the stack. For a native PowerPC application, the two parameters are simply specified as standard C parameters and the following routine suffices (the TEPtr parameter isn't used in this example):

```
void HighHookUnderline (Rect *boundsRect, TEPtr pTE)
{
   boundsRect->top = boundsRect->bottom - 1;
   InvertRect(boundsRect);
   return;
}
```

PASCAL FUNCTIONS

Although the compilers for PowerPC C code were extended to accept the **pascal** keyword for source code compatibility with 680x0 Macintosh code, when the compiler encounters this keyword, it does *absolutely nothing*. Unlike MPW C, where the keyword alters parameter ordering and changes how some parameters are passed, the compilers for PowerPC code ignore the **pascal** keyword. In most cases this is not a problem, but there can be some subtle consequences. For example, consider the following Apple event handler:

```
pascal OSErr DoAEAnswer (AppleEvent message, AppleEvent reply,
   long refCon);
```

An Apple event record is larger than four bytes, so in Pascal it's automatically passed by reference. Because DoAEAnswer is declared as a **pascal** function, MPW C handles the parameter in the same way. But the compilers for PowerPC code treat it as a standard C data structure and pass it by value. So if DoAEAnswer were called by the Apple Event Manager, bizarre things would happen.

To be compatible with both types of compilers, you must explicitly make these parameters pointers, as follows:

```
pascal OSErr DoAEAnswer (AppleEvent *message, AppleEvent *reply,
   long refCon);
```

When in doubt, check the new interfaces; they now declare special function pointers of type ProcPtr that specify the correct parameters.

```
typedef pascal OSErr (*EventHandlerProcPtr)(const AppleEvent
   *theAppleEvent, const AppleEvent *reply, long handleRefCon);
```

Unfortunately, in most cases you'll now be coercing any special ProcPtrs (such as EventHandlerProcPtr) into normal ProcPtrs for calls to NewRoutineDescriptor (described in the next section), which means type checking will be lost. So double-check all your callback routines.

UNIVERSALPROCPTRS

Because of the introduction of the Mixed Mode Manager, the single biggest change you'll have to make to your code is converting function pointers of type ProcPtr to type UniversalProcPtr. Every place in the interfaces where a type of ProcPtr was declared, Apple added a similar declaration of type UniversalProcPtr.

UniversalProcPtr is a generic function pointer. For 680x0 code, a UniversalProcPtr is just a 680x0 ProcPtr. For native PowerPC code, though, a UniversalProcPtr is a pointer to a data structure called a *routine descriptor*, which in addition to providing a function reference, supplies all the information the Mixed Mode Manager needs to transform parameters back and forth between 680x0 and PowerPC worlds. Because a UniversalProcPtr is no longer a simple function reference, there are issues of allocation and scope that make it more complicated to use than a simple ProcPtr. Fortunately, 680x0 interfaces are being changed to add UniversalProcPtr support, so changes you make for PowerPC code will also be compatible with 680x0 interfaces (see "Universal Interfaces" earlier in this article).

Let's look at a simple example using a UniversalProcPtr. Suppose you have an action procedure for a vertical scroll bar, called VActionProc. Current code would call TrackControl with that action procedure as follows:

```
TrackControl(ctlHit, mouseLoc, VActionProc);
```

With PowerPC code, you must create a routine descriptor for VActionProc. Because there's usually a one-to-one correspondence between function pointers of type ProcPtr in your code and function pointers of type UniversalProcPtr required by the Mixed Mode Manager, it's simplest to allocate one UniversalProcPtr for each ProcPtr you use. The memory impact of this approach is small because a routine descriptor data structure typically uses only 32 bytes.

One way to do this is to allocate the routine descriptor statically and have it initialized by the compiler. Macros are supplied in MixedMode.h for this purpose. For example, you can create a routine descriptor for VActionProc like this:

```
RoutineDescriptor gVActionProcRD =
   BUILD_ROUTINE_DESCRIPTOR(uppControlActionProcInfo, VActionProc);
```

Alternatively, you can allocate your routine descriptors on the heap. Again, because they seldom change, you'll generally want to allocate them at application startup:

```
ControlActionUPP  gVActionUPP;

gVActionUPP = NewRoutineDescriptor((ProcPtr)VActionProc,
   uppControlActionProcInfo, GetCurrentISA());
```

NewRoutineDescriptor is declared as follows:

```
UniversalProcPtr NewRoutineDescriptor(ProcPtr theProc,
   ProcInfoType theProcInfo, ISAType theISA);
```

NewRoutineDescriptor allocates nonrelocatable storage for the routine descriptor on the heap and returns it as a pointer to the routine descriptor in the form of a UniversalProcPtr. The theProc parameter is just the function pointer for the function you're referring to and theProcInfo is a 32-bit value that tells the Mixed Mode Manager how to convert parameters back and forth. Every UniversalProcPtr type has defined for it a corresponding ProcInfoType value. So the ProcInfoType value for ControlActionUPP is uppControlActionProcInfo. The third parameter, theISA, specifies the current instruction set architecture (ISA) in use. For portable code, simply call GetCurrentISA to get the appropriate ISA type. If you know you're dealing with a specific code type — for example, a 680x0 code resource — you can call NewRoutineDescriptor and specify the proper instruction set type — for example, kM68kISA for 680x0 code.

To simplify creation of function pointers of type UniversalProcPtr, the new interfaces also define macros that call NewRoutineDescriptor for you and automatically specify the ProcInfoType value:

```
gVActionUPP = NewControlActionProc((ProcPtr) VActionProc);
```

If you created the routine descriptor statically, you can pass the address of the structure to TrackControl:

```
TrackControl(ctlHit, mouseLoc, (ControlActionUPP) &gVActionProcRD);
```

If, instead, you created a UniversalProcPtr on the heap, you can use it directly in TrackControl:

```
TrackControl(ctlHit, mouseLoc, gVActionUPP);
```

If you allocate a UniversalProcPtr statically, you don't have to worry about deallocating it, because that will happen when the application quits. You could also allocate it locally, which you might want to do if the routine were unlikely to be

called. In that case, you would have to explicitly deallocate the routine descriptor before leaving the function, as follows:

```
DisposeRoutineDescriptor(gVActionUPP);
```

A potential problem with disposing of routine descriptors is that you could dispose of them before they're used. For example, if you have a routine descriptor for an asynchronous I/O completion routine, disposing of the routine descriptor before the completion routine is called would be bad.

An alternative for infrequently used routine descriptors is to allocate them globally but initialize them only when needed, as in this example:

```
if (!gVActionUPP)
   gVActionUPP = NewControlActionProc((ProcPtr) VActionProc);
TrackControl(ctlHit, mouseLoc, gVActionUPP);
```

In most cases you won't need to call a UniversalProcPtr yourself; you'll simply pass it to the Toolbox. But should you need to call one from PowerPC code, you can't simply treat it as a function pointer. You must use CallUniversalProc to have the Mixed Mode Manager call the function for you. CallUniversalProc is declared as follows:

```
long CallUniversalProc(UniversalProcPtr theProcPtr,
   ProcInfoType procInfo, ...);
```

The first two parameters, the UniversalProcPtr and the 32-bit ProcInfoType value, are followed by all the additional parameters normally passed to the call. To simplify calling UniversalProcPtrs, special macros have been included in the interfaces for each UniversalProcPtr data type. For example, gVActionUPP above could be called using CallControlActionProc:

```
CallControlActionProc(gVActionUPP, theControl, partCode);
```

One special case of a UniversalProcPtr deserves mention because it can't be flagged by the compiler. A wonderful feature of the Dialog Manager is that for a userItem, the SetDItem call allows the item's procedure pointer to be set via the item parameter. Since you're explicitly casting a ProcPtr to a handle, the compiler assumes you know what you're doing and doesn't object. Of course, what you really need to pass is a UniversalProcPtr, but since the compiler doesn't catch this, strange things will surely happen if you don't catch it yourself.

As another example of using function pointers of type UniversalProcPtr, let's look at a VBL task. A persistent VBL task (one that works when the application is in the background) is often implemented by copying the VBL task code into the system

heap, an ugly solution and self-modifying code as well. A simpler solution for PowerPC code is to create the UniversalProcPtr itself in the system heap since the Process Manager views the UniversalProcPtr as code. The following code shows how to install such a VBL task:

```
#define kVBLInterval 30

OSErr InstallVBL (VBLTaskPtr theVBLTask, VBLProcPtr myVBLProc,
   Boolean isPersistent)
{
   OSErr theError;
   THz   savedZone;

   /  *
      *  For a VBL task that operates when the application is in the
      *  background (i.e., that's persistent) we can simply create the
      *  UniversalProcPtr in the system heap. This causes the Process
      *  Manager to treat the code as though it were in the system heap
      *  and the VBL will always get executed.
      *  /

   if (isPersistent) {
      savedZone = GetZone();
      SetZone(SystemZone());
   }
   theVBLTask->vblAddr = NewRoutineDescriptor((ProcPtr) myVBLProc,
      uppVBLProcInfo, GetCurrentISA());
   theError = MemError();
   if (isPersistent)
      SetZone(savedZone);     /* Restore the application zone. */
   if (theVBLTask->vblAddr != nil) {
      theVBLTask->qType = vType;
      theVBLTask->vblCount = kVBLInterval;
      theVBLTask->vblPhase = 0;
      theError = VInstall((QElemPtr) theVBLTask);
   }
   return (theError);
}
```

The isPersistent Boolean variable controls whether the VBL functions persistently. If it's persistent, you can control where the memory is allocated by first setting the zone to the system zone (because NewRoutineDescriptor calls the Memory Manager to allocate memory for the routine descriptor).

Here's the code for the VBL task:

```
long  gCounter = 0;

pascal void MyVBLProc (VBLTaskPtr theVBLTask)
{
   theVBLTask->vblCount = kVBLInterval;
   gCounter++;
   return;
}
```

This very simple example alters only a global variable, but it illustrates two points. First, no complicated setup for global variables is required. For a 680x0 VBL task, messy saving and restoring of register A5 would be necessary for correct access to global variables. In the example, because the code resides in a code fragment, global variables are always accessible. Second, the procedure is called with a VBLTaskPtr parameter. For a 680x0 VBL task, a pointer to the VBLTask record resides in register A0 and requires special handling to get to the data from a high-level language. Because PowerPC code uses strict C calling conventions, the required data is passed as a standard parameter.

Finally, of course, you have to remove the VBL task correctly:

```
void RemoveVBL (VBLTaskPtr theVBLTask)
{
   THz   savedZone;

   VRemove((QElemPtr) theVBLTask);
   if (theVBLTask->vblAddr) {
      savedZone = GetZone();
      /* Make sure we're in the right zone. */
      SetZone(PtrZone((Ptr) theVBLTask->vblAddr));
      DisposeRoutineDescriptor(theVBLTask->vblAddr);
      SetZone(savedZone);
   }
   return;
}
```

Although it may not be necessary to deallocate a VBL task created in the application heap, this code practices safe memory management by being sure the memory gets deallocated no matter where it is — in other words, whether it's persistent or not.

TRAP PATCHING

Trap patching is fully supported on the PowerPC microprocessor; as always, however, it must be undertaken with due care and consideration. Not only is the compatibility risk higher (especially if you're dependent on 680x0 runtime features), but

indiscriminate trap patching can severely affect the performance of the PowerPC processor–based machine.

Trap patching is possible from both 680x0 code and PowerPC code, and you should use the NGetTrapAddress and NSetTrapAddress calls in both cases. From PowerPC code, the address returned by NGetTrapAddress must be treated as a UniversalProcPtr and you must pass a UniversalProcPtr to NSetTrapAddress as well.

What complicates the issue is that the trap you patch could be written in either 680x0 code or PowerPC code. The Mixed Mode Manager, of course, handles both cases, but if you're patching native PowerPC code with 680x0 code, performance-sensitive code can suddenly run more slowly, not only because of your emulated code but because of overhead associated with mixed mode transitions. So you must think very carefully about the performance consequences of your patch.

TAKING A RISC

To ease the transformation of existing applications into native PowerPC applications, Apple has minimized changes to the API. Most ANSI C compliant code, with the exception of ProcPtrs, should recompile without modification. Developers can exploit this opportunity to easily tap into the power of the PowerPC microprocessor.

With PowerPC processor–based machines, Apple is laying the foundation for the future. The new levels of performance and new features such as the Code Fragment Manager give developers new worlds to explore and new opportunities for adding unique features to their applications.

RECOMMENDED READING

For more information on CISC and RISC architectures in general and POWER and PowerPC architectures in particular, consult the following sources:

- *Advanced Microprocessors* by Daniel Tabak (McGraw-Hill, 1991).
- *Computer Architecture* and *Computer Architecture Case Studies* by Robert J. Baron and Lee Higbie (Addison-Wesley, 1992).
- *Computer Architecture: A Quantitative Approach* by David A. Patterson and John L. Hennessy (Morgan Kaufman Publishers, 1990).
- "PowerPC Performs for Less," by Tom Thompson, *Byte*, August 1993.
- "RISC Drives PowerPC," by Bob Ryan, *Byte*, August 1993.
- *PowerPC 601 RISC Microprocessor User's Manual* (Motorola, 1993).

THANKS TO OUR TECHNICAL REVIEWERS
C. K. Haun, Ron Hochsprung, Bruce Jones, Alan Lillich, Wayne Meretzky, Eric Traut•

SOMEWHERE IN QUICKTIME

WHAT'S NEW WITH SOUND MANAGER 3.0

JIM REEKES

Sound Manager 3.0, a vastly improved call-for-call replacement for the Sound Manager in System 7, provides QuickTime and other sound clients with a set of new and improved features, including higher frame rates and better quality sound. Sound Manager 3.0 is an extension that entirely replaces the older Sound Manager; the extension is included along with a new Sound control panel on this issue's CD. We released it as an extension without changing the API so that you won't have to recode your existing applications. With Sound Manager 3.0 installed in your system, your applications will transparently take advantage of the Sound Manager's greater dependability, speed, and other new features.

The soon-to-be-available *Inside Macintosh: Sound* (or *Inside Macintosh* Volume VI, Chapter 22) is the main source of Sound Manager documentation. This column will discuss some of the new features of Sound Manager 3.0 and describe how to use them.

OVERVIEW OF MAJOR NEW FEATURES

Sound Manager 3.0 provides four major new features:

- support for 16-bit audio samples
- support for third-party audio hardware
- support for plug-in audio codecs
- better performance and quality

Previous versions of the Sound Manager could only support stereo 8-bit audio samples with sample rates up to 22 kHz. Sound Manager 3.0 removes this limitation by allowing stereo 16-bit audio samples with sample rates up to 65 kHz, providing CD-quality audio in QuickTime movies and other audio applications. Sound Manager 3.0 will also automatically convert 16-bit sounds into 8-bit sounds on Macintosh computers that don't have 16-bit audio hardware.

Third-party sound cards can be installed in your Macintosh to allow playback and recording of CD-quality audio. Sound Manager 3.0 makes this possible by providing a driver mechanism and a new Sound control panel that allows the user to redirect sound to any available audio device. Audio card developers can license the Sound Manager 3.0 extension and bundle it for distribution with their product.

The Sound Manager previously supported only MACE audio compression at ratios of 3:1 and 6:1. Sound Manager 3.0 goes beyond MACE to support any compressed audio format with the use of plug-in audio compression/decompression software (codecs). These are simply extension files that the Sound Manager recognizes and uses when it needs to play a compressed sound. In this way, applications can play compressed sounds seamlessly without being aware of the compressed format.

Sound Manager 3.0 is much faster — in many cases two to three times more efficient than previous versions. This means that your application can do more while sound is playing. Sound Manager 3.0 is also more robust: many bugs have been fixed and a number of commonly requested features have been added.

SYSTEM REQUIREMENTS AND INSTALLATION

Sound Manager 3.0 requires the Component Manager, so you must have either System 7 with QuickTime or System 7.1. (The Component Manager comes with QuickTime and is built into System 7.1.) Sound Manager 3.0 supports all Macintosh models except for the "classic"-style hardware such as the Macintosh Plus, SE, and Classic.

JIM REEKES studied music composition and theory in college, never taking a single computer science or engineering class because he knew they would pollute his brain. He taught himself programming, beginning with the Apple II and then on the Macintosh 128K in 1984. He began working in Apple's Developer Technical Support group in 1988. He took over responsibility for the Sound Manager during System 7 beta (so you can't blame that one on him!) and recently finished Sound Manager 3.0, a complete rewrite. If there's one thing he has learned while at Apple, it's that there's a fine line between amazing insight and having a bad attitude. Jim has been collecting progressive rock and electronic music recordings since the 1970s. He grew up in Pomona, California, during the 1960s and can remember when Frank Zappa performed in local bars on Mission Blvd. and Cucamonga was a vineyard. He wishes programming didn't burn out his creative drive so that he could spend more time in his MIDI studio.•

Installing Sound Manager 3.0 consists of dragging the Sound Manager extension and the new Sound control panel to your System Folder (where they will be placed in the appropriate folders) and rebooting. You should see Sound Manager 3.0's icon during startup.

WHAT'S NEW AND IMPROVED

Here are some more details about new and improved features in Sound Manager 3.0.

Speed optimizations. While Sound Manager 3.0 can play virtually any type of sound, it has been optimized for maximum playback efficiency with a number of common sound formats. So if you're worried about performance and want to minimize Sound Manager overhead, use one of these sound formats:

- 8-bit, mono, 22.254 kHz, full volume
- 8-bit, mono, 11.127 kHz, full volume

Increased efficiency is a major improvement in Sound Manager 3.0. In many cases, the Sound Manager will be two to three times more efficient, which allows applications to play more simultaneous sounds and do other work while sound is playing.

For example, you can now play four channels of sound on a Macintosh LC, whereas in the past the Sound Manager would not allow this. QuickTime applications benefit from Sound Manager 3.0 by gaining an increase in the movie playback frame rate. The premiere multimedia platform is now QuickTime 1.6 and Sound Manager 3.0 on a Macintosh!

Sound quality. Sound Manager 3.0 uses a fast linear interpolation for 11 kHz to 22 kHz sample rate conversion, which makes audio sampled at 11 kHz sound much better. This improves the sound quality of many QuickTime movies without sacrificing performance.

16-bit sound. Sound Manager 3.0 includes full support for 16-bit audio samples, including rate conversion, mixing, and decompression. It will automatically convert between 16-bit and 8-bit samples, so you never have to worry about the hardware you're running on. If your system has a 16-bit sound output device, you'll notice an increase in sound quality.

Until now, the value of the sampleSize field of the extended or compressed sound header has been 8 to denote the number of bits per sample. To play 16-bit sounds, specify the value 16 for the sampleSize field in the header, and the Sound Manager will treat the sound data as 16 bits per sample. 16-bit sounds are always in two's complement (signed) representation while 8-bit sounds are always in offset binary (unsigned) representation. For an example of how to fill out the extended sound header so that you can play 16-bit sounds, see Play16BitSound on this issue's CD.

Playing compressed sounds. With Sound Manager 3.0, you can play sounds compressed with any algorithm when you use the CmpSoundHeader data structure. The CmpSoundHeader's old futureUse1 field is now the format field, which you can use to specify a 4-character OSType that identifies the compression algorithm. If the compressionID field of the CmpSoundHeader is set to the constant fixedCompression, the Sound Manager uses the OSType in the format field to find a codec that can decompress this type of audio. The example named PlayCompressedSound on the CD shows how to fill out the compressed sound header so that you can play compressed sounds.

The SndPlayDoubleBuffer call has a similar interface. It accepts a new SndDoubleBufferHeader data structure that's identical to the previous one with the addition of a format field at the end. If the dbhCompressionID field is set to the constant fixedCompression, the format field is used to determine the codec to use to decompress the sound. Otherwise it will work as before.

Multiple sound channels. The overall sound volume (amplitude) has been improved when multiple sound channels are being mixed. In the past the Sound Manager would average the amplitudes for all playing

Sound Manager 3.0 was developed by Jim Reekes and Kip Olson. Kip wants everyone to know that the original design document describing the Sound Manager back in 1987 was titled "Software Architecture for a Device-Independent Sound Manager," which can be abbreviated as SADISM. This explains a lot, doesn't it?•

Sound Manager 3.0 has been made widely available. The extension, control panel, and related files are not only on this issue's CD but are also included in the Sound Manager Developer's Kit v. 3.0 available from APDA, in Hardware System Update 2.0, with sound products from third parties, and on various electronic bulletin boards (such as CompuServe and America Online). Sound Manager 3.0 is built into some new Macintosh systems; you can tell it's there if Sound Out is listed in the Sound control panel.•

channels. With Sound Manager 3.0, this averaging does not occur, which gives you better individual volume control. One possible disadvantage to this is that clipping can occur when many sounds of high amplitude are used.

For those of you trying to synchronize multiple channels, syncCmd could never synchronize at a fine enough level. With Sound Manager 3.0, syncCmd synchronizes multiple channels so that independent sounds can be triggered at exactly the same time. The technique to synchronize multiple channels remains the same as before. See the PlayTwoSoundsSynched example on the CD.

Finding the sound header in a 'snd ' resource. The 'snd ' resource is a cumbersome structure to parse. The old routine SetupSndHeader can be used to create this resource. A new routine, GetSoundHeaderOffset, has been created to locate the embedded sound header, which is used with the soundCmd or bufferCmd. The resulting offset is the number of bytes into the handle to the starting point of the sound header. The handle doesn't have to be locked to get this offset. See the PlaySndHandle example on the CD.

Volume control. Two new sound commands, volumeCmd and getVolumeCmd, allow better control of a channel's output volume. You can use volumeCmd to set the volume. The param2 portion of the command contains a two-word value (four bytes) that represents a pair of volume levels; the high word is the level for the right output signal and the low word is the level for the left. A value of 0x0100 is full volume and 0x0080 is half volume. For an example of setting the volume, see ChangeVolume on the CD.

You can overdrive the volume if you want to amplify low signals. A value of 0x0200 would be twice full volume. Furthermore, you can independently control the right and left volumes. The value 0x01000000 would send the output signal to the right, and 0x00000100 would send it left. The value 0x00800100 would play out the right side at half volume and the left at full volume.

The getVolumeCmd command returns the current volume. The param2 field should be a pointer to a long, similar to getAmpCmd.

There are two new routines for controlling the volume of system beep sounds: GetSysBeepVolume and SetSysBeepVolume.

```
pascal OSErr GetSysBeepVolume(long *level)
 = {0x203C,0x0224,0x0018,0xA800};
pascal OSErr SetSysBeepVolume(long level)
 = {0x203C,0x0228,0x0018,0xA800};
```

SysBeep will create a sound channel adjusted to the volume level last set by SetSysBeepVolume. This allows for system beep sounds to play back at a lower level than the rest of the machine, so you can hear a QuickTime movie running at full volume but hear alert beeps at a softer level.

The older routines GetSoundVol and SetSoundVol were implemented as a Control call to the Sound Driver. Although we've made every effort to continue supporting them, they do not have the amount of accuracy that's available with two new Sound Manager routines GetDefaultOutputVolume and SetDefaultOutputVolume:

```
pascal OSErr GetDefaultOutputVolume(long *level)
 = {0x203C,0x022C,0x0018,0xA800};
pascal OSErr SetDefaultOutputVolume(long level)
 = {0x203C,0x0230,0x0018,0xA800};
```

The older routines used a 0-7 value range whereas the new Sound Manager has a 0-0x0100 range. These new routines use the right/left volume pair as described above for volumeCmd. Each device has its own volume level. If the user changes the selected default device from the Sound control panel, that new device will use its own volume level, originally set by a previous call to SetDefaultOutputVolume.

Better stereo support. Previous versions of the Sound Manager would drop the right channel of a stereo sound when playing on monophonic hardware,

such as a Macintosh LC. Sound Manager 3.0 will automatically convert stereo sounds to mono on these machines without dropping the right channel, so you can hear what you've been missing. Certain older Macintosh models are also mono out of the internal speaker, but stereo if headphones are plugged in. Sound Manager 3.0 will automatically sense if a headphone is plugged in and do the correct conversion so that both the right and left channels of a stereo sound will always be heard. The only exception is the Macintosh IIfx, which requires you to manually select stereo or mono in the new Sound control panel.

Default output device. Sound Manager 3.0 includes the concept of a default output device, set by the user in the new Sound control panel using the Sound Out panel. All sounds will be sent to this device unless an optional device was specified with SndNewChannel. The default device is generally the built-in sound hardware. The user can choose a new device (such as a sound card the user installed), and all sounds will then be routed to the chosen device. Adjusting the volume with either the control panel or the older call to SetSoundVol adjusts the volume of the default device.

Integration with QuickTime. QuickTime 1.6 is aware of Sound Manager 3.0 and will take advantage of its new features if it's installed.

- Option-clicking the volume control in QuickTime's movie controller allows you to overdrive the volume of the movie, giving a boost to low signals.
- The track balance of an audio track can now be proportionally panned left and right, instead of just full left or full right.
- QuickTime will query Sound Manager 3.0 for information on new compression types, allowing it to play compressed audio of any type. It will send 16-bit audio data directly to the Sound Manager, so QuickTime movies can play CD-quality audio.
- QuickTime will use the Sound Manager to do rate conversion and mix multiple sound tracks into one sound for export as an AIFF file or 'snd ' resource.

Sound Driver compatibility. The old Sound Driver, including the use of SoundBase, still works with Sound Manager 3.0, but we don't know how much longer this will be true. This depends entirely on changes in the hardware, not on the Sound Manager. If you're currently using the Sound Driver, Apple strongly encourages you to use the Sound Manager instead. Future changes in the sound architecture will be transparent to your application if you use the Sound Manager; they won't be if you continue to use the Sound Driver.

CPU loading. The Sound Manager released with system software versions 6.0.7 and later contained support for CPU loading. This approach was found not to be very accurate, and is not supported in Sound Manager 3.0. Sound Manager 3.0 will return the constant 7% for any channel, no matter how it was created and initialized. The number 7% was chosen because some applications were expecting a nonzero value, and 7% is about right for a Macintosh LC playing a single 11 kHz mono sound. Since the Sound Manager doesn't have true CPU loading checks, it's possible to run out of real time and thus overload the machine. Sound will then break up or even hang the system. This problem will be addressed in a future version of the Sound Manager.

Synth modes. Previously the Sound Manager enforced a single synthesizer type to be allocated. Even if a given synthesizer type allowed for multiple channels, you still couldn't mix the types. For example, you couldn't use the wave table mode while any other mode was operating. This limitation has been eliminated. Any and all three types of channels (square, wave table, and sampled sound) can be opened and used at the same time.

Square wave sounds. Unknown to most, the square wave synthesizer never produced true square waves. It was more like a modified sine wave. This has been corrected. As a result you'll notice that the Simple Beep sounds different. It can now be heard as it was originally designed to sound.

BUG FIXES AND FEATURE ENHANCEMENTS

The following is a brief summary of bugs that have been fixed in Sound Manager 3.0. This is not a complete list. Its intention is to point out major areas of improvement that might affect a large number of applications.

Play from disk

- Some asynchronous file I/O problems while operating under the asynchronous SCSI Manager have been fixed.
- Incorrect calculation of the audio selection for anything other than noncompressed 8-bit sounds has been fixed. This makes MACE and 16-bit data work with selections.
- SndStartFilePlay can now handle 16-bit sounds and any compressed format.

Sound Input Manager

- Sample rates greater than 32 kHz, which used to create overflows of the Fixed type and produce negative results, are now allowed.
- Record to disk works better with large file system caches. Previously, during long disk writes to flush the file system's cache, incoming sound data would occasionally be lost.
- When opening a sound input driver, the Sound Input Manager now checks for errors returned from the driver.

Sound Output Manager

- Sample rates greater than 32 kHz, which used to create overflows of the Fixed type and produce negative results, are now allowed.
- MoveHHi has been patched to avoid stack-into-heap problems during sound interrupts.
- There are fewer clicks and pops, especially when opening a sound channel.
- When playing multiple channels of sound using the bufferCmd, the Sound Manager will no longer mix in random amounts of silence, which caused sounds to be discontinuous and get out of sync.
- Stopping a sound or starting a new one sometimes caused the channel to fail to produce any new sounds. This has been fixed.
- The ampCmd works for all types of sound channels (square, wave table, and sampled).
- Loop points now work on any type of sound, including 16-bit, stereo, and compressed sounds.
- Linear interpolation is now performed across separate buffers, so you can play a set of sounds without getting a click between sounds.
- Machines with the Macintosh II ROM (II, IIx, IIcx, SE/30) could lose sound interrupts after playing for long periods of time. This has been fixed.

SOUNDING OFF

Sound Manager 3.0 is a vast improvement over the old Sound Manager and will enhance QuickTime applications and other applications that use sound. So check it out; from the system beep to sophisticated movies, we're sure you'll notice the difference.

ADVANCED FEATURES

An important feature of Sound Manager 3.0 is the ability to play through alternate sound output devices installed in your system. These devices will be available from third-party developers. The Sound Manager can take advantage of specialized hardware features such as sample rate conversion and audio mixing. If such features are available in the hardware (such as better sample rate conversion done by a DSP), the Sound Manager will allow this support to be passed off to the hardware for better quality and efficiency.

Support for plug-in audio codecs is another significant new feature. This allows the Sound Manager to support new compression methods, which become desirable now that we're supporting 16-bit data.

Thanks to Ray Chiang and Bryan ("Beaker") Ressler for reviewing this column. Special thanks to Kip Olson.•

BUILDING POWERTALK-SAVVY APPLICATIONS

PowerTalk is a new software product based on the Apple Open Collaboration Environment (AOCE). By adding support for PowerTalk to your application, you can begin to take advantage of the wide range of services provided by the emerging world of collaborative computing. This article touches on two areas of this environment — electronic mail and digital signatures — and shows how they can be incorporated into a typical application program.

STEVE FALKENBURG

AOCE consists of a set of human interface elements and programming interfaces that make collaboration on an electronic document simpler and more secure; PowerTalk is its client software component (and PowerShare its server software). Two elements of PowerTalk are the Standard Mail Package's mailer, which provides application-level electronic mail support, and the DigiSign digital signature mechanism, which safeguards documents from electronic tampering. Support for these features of PowerTalk should not be limited to networking and communications applications. The real power of PowerTalk lies in its ability to be built into a wide range of productivity applications, from spreadsheets to presentation packages. Ultimately, the Send and Sign menu items should be as pervasive as Print is today.

Using a small drawing application called CollaboDraw as our example, we'll go step by step through the process of adding support for the PowerTalk Standard Mail Package and Digital Signature Package.

WHAT EVERY APPLICATION SHOULD KNOW ABOUT POWERTALK

Before walking through the code, we'll give a brief overview of the PowerTalk features we'll be adding to CollaboDraw. Very basic descriptions of the Standard Mail Package and Digital Signature Package follow. Additional information on PowerTalk can be found in the full PowerTalk API documentation.

STEVE FALKENBURG has been working in Apple's Developer Technical Support group ever since he finished his last *develop* article nearly three years ago. When not supporting PowerTalk (and Macintosh on PowerPC), Steve can be found hiking around California everywhere from Mount Tamalpais to Big Basin. Some people think he's just searching for the perfect mountain vista, but he's also trying his best to keep pace with his hiking partner, Nancy.•

PUSHING THE STANDARD MAIL ENVELOPE

One of the unique features of PowerTalk is that it allows many individual applications to add support for mailing documents directly, without going through an intermediate e-mail application such as QuickMail or AppleLink. The Standard Mail Package provides a consistent interface for mailing documents from one user to another within applications, and includes all of the human interface elements needed to address, send, and receive messages. The major component of the Standard Mail Package is the *mailer*. The mailer is a window pane that's at the top of all documents that are mailed. The mailer window pane can be contracted to display only a single line or expanded to allow manipulation of the mailer's contents. Figure 1 shows a CollaboDraw window containing an expanded mailer window pane.

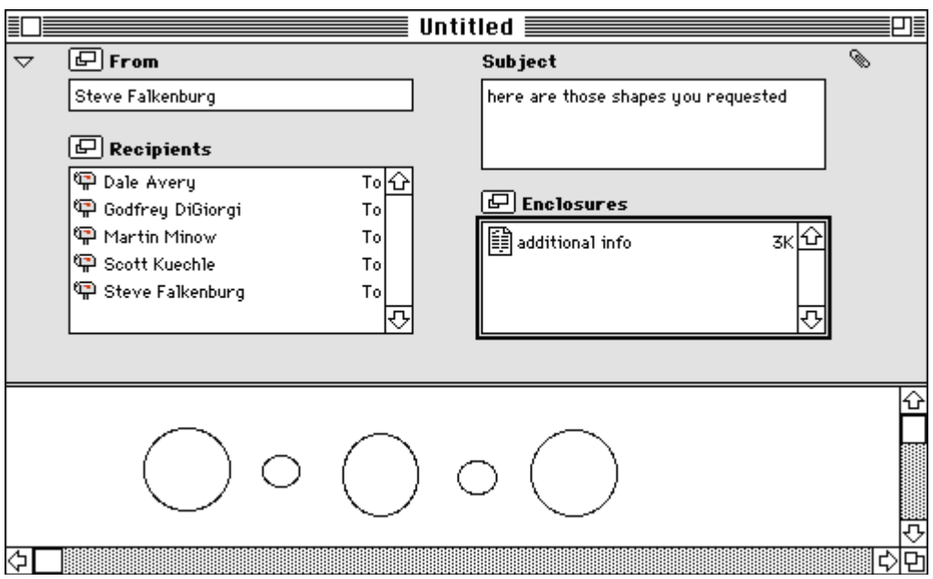


Figure 1
CollaboDraw Window With a Mailer

The mailer can be thought of as a kind of extended mailing label. It contains not only the names of the sender and receivers of the letter, but also a subject for the letter and an area where files and folders can be enclosed.

Making an application mail-aware involves adding several standard menu items. In CollaboDraw, there's a separate Mail menu, but if this isn't a viable option, it's acceptable to add these menu items to the File menu. The standard Mail menu is shown in Figure 2. The items Reply to All, Open Next Letter, and Tag Letter are optional and not required for minimal mailer support.

Figure 2
The Mail Menu

When users want to send a document from CollaboDraw, their favorite PowerTalk-savvy drawing program, they simply add a mailer to their drawing document, transforming the document into a letter. They fill out the mailer and choose Send from the Mail menu. The letter is then sent automatically to the recipient's mailbox in the Finder. Recipients of the document would, in turn, double-click the letter they received in their PowerTalk Finder mailbox, which opens the letter in their copy of CollaboDraw and displays it with the attached mailer. Once they were done reviewing the letter, they could keep the mailer attached if they wanted to reply to the letter, forward the letter, or keep the additional information the mailer provides. Or they could select Remove Mailer from the Mail menu, which removes the mailer from the window, transforming the letter back into a document. (The Remove Mailer menu item replaces Add Mailer when there's a mailer in the window.)

Much of the power of PowerTalk Standard Mail stems from the fact that all PowerTalk-aware applications support an additional file type: the letter. In the Finder, letters can appear in disk windows, in the PowerTalk Finder mailbox window, and even on the desktop or in the trash. Users can treat these letters like standard documents, dragging them between folders to copy them, dragging them to the trash to erase them, and even double-clicking them or dragging them to an application to open them. When integrating mailer support into an existing application, it's best to think of letters in much the same way — simply as an additional document type. Using this strategy, we'll see that adding a mailer requires little in the way of application redesign.

UNLOCKING THE POWER OF DIGITAL SIGNATURES

Another very powerful PowerTalk feature that can be added to document-based applications with a small amount of effort is digital signatures. PowerTalk's DigiSign digital signature technology allows you to apply a personal "signer" to an object or a file before distribution. Other users can then verify the digital signature, which guarantees the identity of the person who signed the object as well as ensuring that

the object has not been altered in any way. If the object is modified after being signed, the signature verification will fail, which will indicate that either the object has changed or the signature has been tampered with.

Digital signature support also requires adding several menu items. These items are normally added to an application's Edit menu, but because CollaboDraw has plenty of space in the menu bar, they were separated into a Signatures menu (see Figure 3).

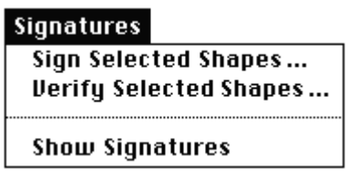


Figure 3
The Signatures Menu

Within CollaboDraw, digital signature support is provided for the individual shapes and groups of shapes. To sign a shape, the user simply selects the shape (or group) and then chooses Sign Selected Shapes from the Signatures menu. A dialog box appears, prompting for the user's signer identification code. Once the user enters the password protecting the signer, the selected shape is signed; a dashed rectangle appears around the shape, with a small icon button (labeled with a pen) in the lower right corner, indicating that the shape has been signed. (If you were adding digital signature support to a text-based application, the dashed rectangle would surround the signed text.) Figure 4 shows a signed shape.

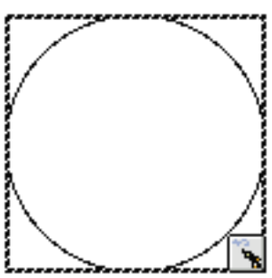


Figure 4
A Signed Shape

To verify the integrity of the signature, a user could either click the pen button in the corner of the shape or select the shape and choose Verify Selected Shapes from the

DigiSign's digital signature implementation is based on a public key/private key standard developed by RSA Technologies, Inc. The technology underlying digital signatures is beyond the scope of this article; interested readers should refer to the PowerTalk documentation for more information.•

Signatures menu. If the signature verification is successful, the dialog box in Figure 5 is displayed, showing the identity of the signer.

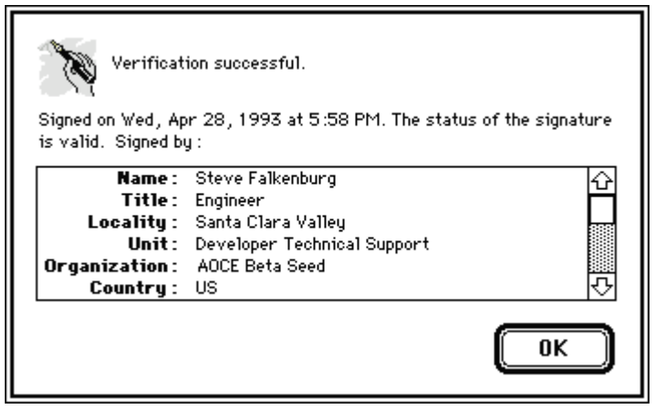


Figure 5
Signature Verification Dialog Box

The DigiSign Digital Signature Manager provides routines to display the dialog boxes described above, as well as standard icons for use in constructing the pen icon button. This makes adding digital signature support a relatively painless operation.

LETTER FORMATS

As was mentioned earlier, you can think of letters as another type of document that your application needs to support. Before describing how to add support for this new document type, we'll spend some time discussing the format of PowerTalk letters.

Letters are a special kind of PowerTalk *message*. A letter is different from a message in that it is sent from one user to another and is meant to be read by a human, whereas a standard message is sent from one program to another and is meant to be read by a program. Both share the same low-level format, consisting of a *message header* and a series of *message blocks*.

The message header describes the message as a whole, including who the message is from, who the message is to, the subject of the message, the date it was sent, and whether the message is a letter. The header stores most of the information contained in the mailer window pane shown in Figure 1, with the exception of enclosures.

Each message also contains message blocks, where the actual message data is stored. Each block has a type and a creator, as well as message data and a length field.

PowerTalk-defined message blocks store message enclosures, digital signatures, or message content. In addition, application-specific message blocks can be stored here.

PowerTalk letters have a well-defined content format, which is made up of any combination of three formats: AppleMail format, Snapshot format, and Native Application format. Figure 6 shows a letter with all three of these content formats.

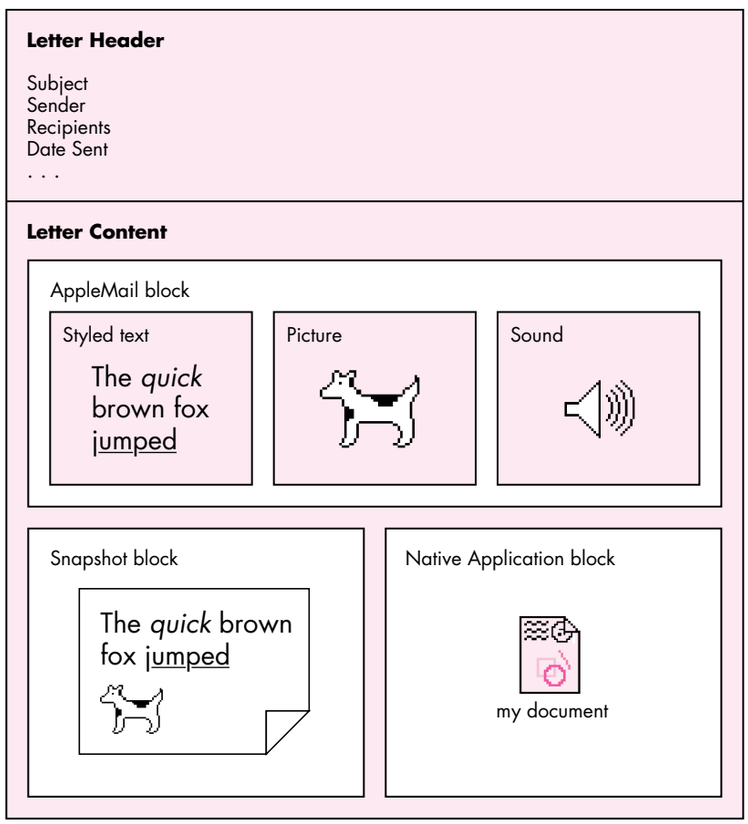


Figure 6
PowerTalk Letter With Content Blocks

AppleMail format is one of the most commonly supported content types. It's made up of runs of text, styled text, PICTs, sounds, and QuickTime movies. In Figure 6, the AppleMail block contains a small amount of styled text, followed by a picture, followed by a sound in AIFF format. Using Standard Mail routines, applications can easily get this content out of a letter and display it to the user. AppleMail format is

the native format for the AppleMail letter application, which ships with PowerTalk. This means that if you send a letter from your mail-aware application and the recipient doesn't have a copy of that application, the recipient will still be able to read the letter's content if you included it in AppleMail format. In addition, PowerTalk Mail Service Access Modules (MSAMs) will most likely use this format to convert messages to other external mail systems.

Snapshot format consists simply of PICT snapshots of each page of your letter. It's similar to AppleMail format in that other letter applications or MSAMs are likely to be able to read mail sent in this format. Snapshot format is provided for the convenience of fax gateways, which can easily use it to image letters to fax machines, and also to offer a WYSIWYG format that preserves the exact look of the original document.

For applications that use QuickDraw GX, the graphics content for each page needs to be translated into standard QuickDraw before it can be added to the Snapshot content block. QuickDraw GX provides a set of routines for this purpose, contained in PicturesAndPICTLibrary (and documented within that library's source code on the QuickDraw GX CD). These routines allow you to pass in a QuickDraw GX picture and receive a QuickDraw PICT as a result. On a QuickDraw system, only the QuickDraw data in this PICT will be drawn. When you pass the PICT into a QuickDraw GX system and convert it with the GXConvertPICTToShape routine, the routine will use the QuickDraw GX data rather than the QuickDraw data.

Finally, Native Application format is basically a copy of your original document's disk file put into the letter's content area. This format, meant mostly for the private use of your application, is useful for sending documents between two users who both have the same PowerTalk-aware application. For example, if two users had CollaboDraw, our mail-aware application, and one user sent the other a CollaboDraw letter that included the document in Native Application format, the receiving application could simply extract an FSSpec for the document file to interpret the data in that document. This means you won't lose information by translating your document into another format, but can instead preserve your private document format.

BUILDING THE FRAMEWORK

It's time to look at our sample application. For simplicity, PowerTalk support will be added to a limited MacDraw®-like application, CollaboDraw, included on this issue's CD. In this section I briefly describe the basic application framework. Later sections will show how I added support for the mailer and digital signatures.

The CollaboDraw application is based on a simplified object-oriented message-passing framework. It's simplified in that only windows are treated as objects, and the code is actually written in C, not C++. The basis for this object scheme is a block containing the window content, along with functions, called *methods*, for processing

events that occur in that window. The block is a handle that's allocated dynamically for each window and is stored in the window's refCon field. In this way, I can remove all of the multiwindow complexity from my event loop and simply send a message to the window receiving the event, letting it take its own action.

I won't go into the details of what the CollaboDraw framework does, as I want to concentrate on the PowerTalk aspect of the sample. It's important to recognize, however, that CollaboDraw is a fairly typical drawing application. As you'll see, it's certainly not necessary to redesign an application to add PowerTalk support.

ADDING STANDARD MAIL PACKAGE SUPPORT

A large part of the PowerTalk support code in CollaboDraw is for the mailer window pane and for enabling the mailer to send and receive letters. I've outlined the necessary code below, with samples interspersed showing proper use of the Standard Mail Package calls.

INITIALIZING STANDARD MAIL

Before using PowerTalk in CollaboDraw, we first need to make sure that PowerTalk services are available. This is done once when CollaboDraw launches. The following routine checks whether PowerTalk is installed and available:

```
Boolean HasStandardMail(void)
{
   OSErr err;
   long  response;

   err = Gestalt(gestaltSMPMailerVersion, &response);
   if ((err!=noErr) || (response==0))
      return false;
   return true;
}
```

The above routine determines whether PowerTalk and the mailer calls are available by checking the gestaltSMPMailerVersion attribute. Since PowerTalk may not be installed or may be disabled, quitting when PowerTalk is unavailable is incorrect behavior. Instead, like CollaboDraw, the application should just disable or hide its PowerTalk services, letting the user work with the rest of the application normally.

Once it's known that PowerTalk is available and active, the next step to using Standard Mail services in CollaboDraw is to initialize the Standard Mail Package.

```
OSErr InitStandardMail(void)
{
   OSErr err;
```

```
   SetCursor(&gWatchCursor);
   err = SMPInitMailer(kSMPVersion);
   SetCursor(&qd.arrow);
   return err;
}
```

SMPInitMailer takes the current version number of the Standard Mail Package as input. Later versions of PowerTalk will continue to support older Standard Mail calls by identifying the version the application was compiled with and mimicking those interfaces.

OPENING AND CREATING A LETTER

Now that CollaboDraw has checked for and initialized the Standard Mail Package, it can continue normally, entering its event loop. The next support code we'll cover deals with opening letters and creating new letters from existing drawings.

Typically, a user opens a letter in CollaboDraw, or any other mail-aware application, by double-clicking a letter in the Finder. This, in turn, generates an Open Document core Apple event, which we process in the normal way, with one change: instead of getting the FSSpec out of the event, mail-aware applications need to check the type of each item in the event, handling both FSSpecs and LetterSpecs. The LetterSpec is necessary since PowerTalk letters, in addition to residing in the file system, can be opened from the PowerTalk mailbox, which is not an HFS volume. A LetterSpec uniquely identifies a letter inside the mailbox and can be passed via an Apple event to the mail-aware application to open a letter. The following section of the Apple event handler shows how to process both LetterSpecs and FSSpecs:

```
AECountItems(&docList, &itemsInList);
for (index=1; index<=itemsInList; index++) {
   err = AESizeOfNthItem(&docList, index, &returnedType, &size);
   if (err!=noErr)
      return err;

   if ((returnedType == typeLetterSpec) || (returnedType==typeFSS)) {
      diskForm = false;
      err = AEGetNthPtr(&docList, index, typeLetterSpec, &keywd,
            &returnedType, (Ptr)&myLetterSpec, sizeof(LetterSpec),
            &actualSize);
   } else if (returnedType == typeAlias) {
      diskForm = true;
      err = AEGetNthPtr(&docList, index, typeFSS, &keywd, &returnedType,
            (Ptr)&myFSS, sizeof(myFSS), &actualSize);
   }
   if (err!=noErr)
      return err;
```

```
   if ((returnedType==typeLetterSpec) || (returnedType==typeAlias) ||
         (returnedType==typeFSS)) {
      err = HandleOpenDoc(diskForm, &myFSS, &myLetterSpec);
      if (err!=noErr)
         return err;
   }
}
```

To handle opening either LetterSpecs or FSSpecs as letters, PowerTalk defines a variant structure called a LetterDescriptor that supports both formats. Once we have a LetterDescriptor, we can use this information to open the letter. The mailer-window method CollaboDraw uses to open letters is shown below.

```
void *DMailerLoadWindow(WindowPtr window, WInfoPtr infoPtr, void *data)
{
   OSErr             err;
   LetterDescriptor  *letterDesc;
   Point             upLeft = {0, 0};
   FSSpec            enclSpec;
   Handle            letterDescHndl;

   . . .
   letterDesc = (LetterDescriptor *)data;
   . . .

   err = SMPOpenLetter(letterDesc, window, upLeft, true,
         gPreferences.expandOnOpen, nil, 0L);   // Open the letter.
   if (err!=noErr) {
      DoError(err);
      return nil;
   }

   err = SMPGetMainEnclosureFSSpec(window, &enclSpec);
   if (err!=noErr) {
      DoError(err);
      return nil;
   }
   return DrawLoadWindow(window, infoPtr, &enclSpec);
}
```

After some housekeeping, which has been omitted for clarity, the load method given above calls SMPOpenLetter to open the letter in an existing window. The window was created earlier and was passed into the load method as input. SMPOpenLetter registers this window with the Standard Mail Package and associates it with the letter identified in the LetterDescriptor. SMPGetMainEnclosureFSSpec is then called to

extract the native CollaboDraw document out of the letter, as described earlier in the section "Letter Formats." Finally, the standard CollaboDraw load method is called, which reads the shapes from the document and draws them in the window. CollaboDraw supports opening only letters that contain its native application format, meaning that if the main enclosure block is not present, CollaboDraw doesn't open the letter. For an application to support opening letters without native application content, translation into one of the other content types would be necessary.

In addition to opening existing letters, CollaboDraw allows users to add mailers to existing documents, transforming these documents into letters. When a user chooses the menu item Add Mailer, the following routine is called:

```
void MakeMailerFromDrawing(WindowPtr window)
{
   WInfoPtr infoPtr;
   char     hState;
   Point    topLeft = {0, 0};
   OSErr    err;
   short    mWidth, contHeight, expHeight;

   SetWindowKind(window, kDrawMailerWindow);
   infoPtr = BeginWindowAccess(window, &hState);
   . . .

   // Add the mailer.
   err = SMPNewMailer(window, topLeft, true, gPreferences.expandOnCreate,
         kDefaultIdentity, nil, 0L);
   if (err!=noErr)
      DoError(err);

   // Set the window indent fields.
   err = SMPGetDimensions(&mWidth, &contHeight, &expHeight);
   if (err!=noErr)
      DoError(err);
   if (infoPtr->otherFlags[kMailerExpanded])
      infoPtr->topIndent = expHeight;
   else
      infoPtr->topIndent = contHeight;
   MoveScrollBars(window);
   EndWindowAccess(window, hState);
}
```

When a user chooses to turn a document into a letter, the MakeMailerFromDrawing routine first changes the class of the window. This in turn causes the mailer-window methods, instead of the draw-window methods, to be called in response to events.

Next, this routine adds a mailer to the window with an SMPNewMailer call. Like SMPOpenLetter, this routine associates a particular window with a Standard Mail letter. The kDefaultIdentity parameter to SMPNewMailer is defined as 0 and indicates that the Standard Mail Package should track identities for the application. Finally, the content area of the window is lowered to account for the added height of the mailer. This height can be obtained with an SMPGetDimensions call, which returns both the expanded and contracted heights of the mailer.

HANDLING EVENTS IN MAILER WINDOWS

Since letters are a new document type, new methods are needed to handle events in letter windows. As we'll see, however, we can leverage off of our window class structure to minimize additional code.

When a window contains a mailer, PowerTalk handles a subset of events for that window automatically. This includes mouse-down events, key-down events, update events for the mailer window pane, activate events, deactivate events, and even null events. The event-handling method for mailer windows is as follows:

```
void *DMailerEventWindow(WindowPtr window, WInfoPtr infoPtr, void *data)
{
   SMPMailerResult   whatHappened;
   EventRecord       *ev;
   OSErr             err;

   ev = (EventRecord *)data;
   err = SMPMailerEvent(ev, &whatHappened, nil, 0L);
   if (err!=noErr)
      DoError(err);
   return (void *)(ProcessPowerTalkWhatHappened(window, infoPtr,
                  whatHappened));
}
```

So that PowerTalk will get a first look at the events, CollaboDraw calls SMPMailerEvent with each event received via WaitNextEvent when the frontmost window is a mailer window. This routine will return a value in the whatHappened field indicating what action Standard Mail took and whether you still need to process the event. Here's the postprocessing code for these events:

```
Boolean ProcessPowerTalkWhatHappened(WindowPtr window, WInfoPtr
                           infoPtr, SMPMailerResult mailResult)
{
   OSErr          err;
   SMPMailerState state;
   long           *lastChanged;
```

```
   // See if mailer has changed since we last changed the mailer menus.
   err = SMPGetMailerState(window, &state);
   if (err != noErr)
      DoError(err);
   lastChanged = (long *)&infoPtr->otherData[kLastChangedData];
   if (*lastChanged != state.changeCount) {
      *lastChanged = state.changeCount;
      infoPtr->changed = true;
      FixMailerMenus(window, infoPtr);
   }
   if ((mailResult & kSMPContractedMask) != 0)
      HandleContract(window, infoPtr);

   if ((mailResult & kSMPExpandedMask) != 0)
      HandleExpand(window, infoPtr);
   if (((mailResult & kSMPMailerBecomesTargetMask) != 0) ||
         ((mailResult & kSMPAppBecomesTargetMask) != 0))
      FixMailerMenus(window, infoPtr);

   // Check the menus for *every* event that the mailer handles.
   // We may need to update the Undo item in the File menu.
   if ((mailResult & kSMPAppShouldIgnoreEventMask) != 0)
      FixMailerMenus(window, infoPtr);
   if ((mailResult & kSMPAppMustHandleEventMask) != 0)
      return false;     // App must handle this event.
   else return true;    // Mailer handled this event completely.
}
```

Most of the postprocessing involves recalculating the menu items, since the mailer may have affected which items should be active. In addition to this menu handling, if the kSMPContracted or kSMPExpanded bit is set as a result of the event, CollaboDraw calls its own private routine HandleExpand or HandleContract. In turn, this routine calls SMPExpandOrContract to expand the mailer to its full size or contract it to a single line.

Besides generic event processing, we need to add some minor modifications to the mouse-click method for mailer windows. This is reasonably straightforward:

```
void *DMailerClickWindow(WindowPtr window, WInfoPtr infoPtr, void *data)
{
   RgnHandle   savedClip;
   GrafPtr     savePort;
   void        *returnVal;
   OSErr       err;
   Boolean     alreadyChanged;
```

```
    // Make sure we can change the letter.
    alreadyChanged = infoPtr->changed;
    if (!alreadyChanged && (gCurrentShape!=kSelectShape)) {
       err = SMPPrepareToChange(window);
       if (err==userCanceledErr)
          return nil;
    }

    // Since we're drawing a shape, clear any mailer undo buffer.
    err = SMPClearUndo(window);
    if (err!=noErr)
       DoError(err);

    // Remove mailer from clipping region.
    GetPort(&savePort);
    SetPort(window);
    savedClip = NewRgn();
    GetClip(savedClip);
    ClipToDrawing(window, infoPtr);

    // Call draw-window click method and maybe mark letter changed.
    returnVal = DrawClickWindow(window, infoPtr, data);
    if (!alreadyChanged && infoPtr->changed) {
       err = SMPContentChanged(window);
       if (err!=noErr)
          DoError(err);
    }

    // Restore clipping region.
    SetClip(savedClip);
    DisposeRgn(savedClip);
    SetPort(savePort);
    return returnVal;
}
```

Before passing the click up to the draw-window method to draw or select shapes, we need to notify PowerTalk that the letter content will be changing. To do this, we first call SMPPrepareToChange. If the letter has been digitally signed as a whole, a dialog box warning the user will appear. If the user cancels the change in response to the dialog box (the user may not want to invalidate the signature), the routine exits. Next, the SMPClearUndo routine clears any undo operations from the mailer undo buffer, since only one undo can be pending for a single window. Then the draw area is removed from the window's clipping region, and the superclass click method is called. Upon return, SMPContentChanged is called if the letter has changed. Finally, the clipping region is restored and the method exits.

As you may have noticed from the above discussion, the mailer keeps its own undo buffer. This is because Standard Mail supports the Clipboard operations of Cut, Copy, Paste, Clear, Select All, and Undo for the mailer portion of letters. The code necessary to support the Clipboard is shown in the mailer-window Cut method:

```
void *DMailerCutWindow(WindowPtr window, WInfoPtr infoPtr, void *data)
{
   #pragma unused (data)
   OSErr            err;
   SMPMailerResult   whatHappened;

   err = SMPMailerEditCommand(window, kSMPCutCommand, &whatHappened);
   if (err!=noErr)
      DoError(err);

   return (void *)(ProcessPowerTalkWhatHappened(window, infoPtr,
                  whatHappened));
}
```

As you can see, support for Clipboard operations involves just a single call to SMPMailerEditCommand followed by a call to the CollaboDraw routine ProcessPowerTalkWhatHappened. Similar methods are used for Copy, Paste, Clear, Select All, and Undo.

SENDING A LETTER

Using the code discussed above, CollaboDraw can open and create letters, as well as address them via the mailer. However, a mail-aware application needs to be able to send letters as well. This section extracts the relevant pieces of the CollaboDraw CommSendLetter routine to explain the process of sending a letter step by step.

The first step in sending a letter is to display the send options dialog box. This dialog is very similar to the standard print dialog, providing the user with options as to how the letter should be sent. CollaboDraw uses the following code to display this dialog:

```
GetResString(nativeFormat, kAppNameID, kAppName);
GetWTitle(window, docTitle);
nativeFormatArray[0] = (StringPtr)nativeFormat;
SetCursor(&qd.arrow);
err = SMPSendOptionsDialog(window, docTitle, nativeFormatArray, 1,
      kSMPNativeMask | kSMPImageMask | kSMPStandardInterchangeMask,
      &gPreferences.sendFormat, nil, 0L, &gPreferences.sendFormat,
      &gPreferences.sendOptions);

if (err==userCanceledError)
   return;
```

```
if (err!=noErr) {
   DoError(err);
   return;
}
```

The SMPSendOptionsDialog routine is built into the Standard Mail Package and handles the task of prompting the user for send options. As input, this routine takes the mailer window, the name of the document being mailed, a list of supported native formats, a list of which send formats are supported, and several other send option flags. This routine returns the name of the format that should be used to send the letter, which is used in the next part of the send process:

```
SetCursor(&gWatchCursor);

// Use our creator if we have native format, else use AppleMail creator.
if ((gPreferences.sendFormat.whichFormats & kSMPNativeMask)!=0) {
   letterCreator = kAppCreator;
   letterType = kCDLtrMsgType;
}
else {
   letterCreator = 'lap2';
   letterType = kMailLtrMsgType;
}
err = SMPBeginSend(window, letterCreator, letterType,
      &gPreferences.sendOptions, &mustAddContent);
if (err!=noErr) {
   SetCursor(&qd.arrow);
   EndWindowAccess(window, hState);
   DoError(err);
   return;
}
if (mustAddContent) {
   if (err==noErr)
      err = AddLetterBlocks(window, infoPtr, &gPreferences.sendFormat);
   if (err!=noErr)
      DoError(err);
}
err = SMPEndSend(window, (err==noErr));
if (err!=noErr)
   DoError(err);
```

The above code first calls SMPBeginSend to start the send process. The send options are passed as input to this routine, and relevant information is extracted to build the header for the letter. This call also signals the Standard Mail Package that any content-adding calls apply to the letter specified in the SMPBeginSend call.

Next, the actual blocks of content are added to the letter with the CollaboDraw AddLetterBlocks call, described below. Note that the content blocks are added only if mustAddContent, which is returned from SMPBeginSend, is true. It isn't necessary to add content blocks if a letter is being forwarded unchanged.

Finally, the SMPEndSend call completes the send process. The second parameter to SMPEndSend is true if the letter should be sent, false if it should be aborted.

The AddLetterBlocks routine described above adds the content in any combination of the three formats described earlier in the section "Letter Formats." It simply checks the sendFormat parameter returned from the SMPSendOptions dialog box to determine which formats to add. Native Application format is specified by kSMPNativeMask, AppleMail format by kSMPStandardInterchangeMask, and Snapshot format by kSMPImageMask.

Routines for adding content in the three formats follow.

Native Application format. The AddNativeContent routine adds content in Native Application format to a letter.

```
OSErr AddNativeContent(WindowPtr window, WInfoPtr infoPtr,
                       StringPtr nativeFormatName)
{
   OSErr          err;
   FSSpec         fSpec;
   OCECreatorType blockType;

   // Save file temporarily.
   err = SaveFileToTemp(infoPtr, &fSpec);
   if (err!=noErr)
      return err;
   err = SMPAddMainEnclosure(window, &fSpec);
   FSpDelete(&fSpec);

   // Add native format name string block.
   if (err==noErr) {
      blockType.msgCreator = kMailAppleMailCreator;
      blockType.msgType = kSMPNativeFormatName;
      err = SMPAddBlock(window, &blockType, false, &nativeFormatName[1],
            nativeFormatName[0], kMailFromStart,0);
   }

   return err;
}
```

Native content is stored and accessed via file system FSSpecs, so adding content in this format requires that the document to be included first be saved in a temporary file. The SaveFileToTemp routine, not shown here, does this. Once an FSSpec to the document is available, SMPAddMainEnclosure is called and passed the letter window and the FSSpec. Finally, once this routine completes, a block is added to indicate the name of the native format used in the letter. Note that the native content for CollaboDraw is simply a CollaboDraw drawing document. This document is extracted when a letter is opened to get the list of shapes present in that letter.

AppleMail format. Content in AppleMail format is added with the following routine:

```
OSErr AddAppleMailLetterContent(WindowPtr window, WInfoPtr infoPtr)
{
   OSErr       err;
   PicHandle   thePicture;

   thePicture = DrawImageToPicture(window, infoPtr);
   if (thePicture) {
      HLock((Handle)thePicture);
      err = SMPAddContent(window, kMailPictSegmentType, false,
            *thePicture, GetHandleSize((Handle)thePicture), nil,
            true, smRoman);
      KillPicture(thePicture);
   }
   else
      return kInternalError;

   return err;
}
```

Content in AppleMail format consists of a series of blocks containing text, styled text, pictures, sound, or movies. For CollaboDraw, we simply add a picture block containing all of the shapes in the current document. To add this block, we first call DrawImageToPicture, a CollaboDraw routine to allocate a PicHandle containing the shapes. We then call SMPAddContent with this picture to add the block.

Snapshot format. The final content format supported by CollaboDraw is Snapshot format, and the routines below add a Snapshot block to a letter.

```
OSErr AddLetterImage(WindowPtr window, WInfoPtr infoPtr)
{
   return SMPImage(window, DrawImageProc, (long)infoPtr, false);
}
```

```
pascal void DrawImageProc(long refCon, Boolean inColor)
{
   #pragma unused (inColor)
   OpenCPicParams newHeader;
   OSErr          err;
   Point          zeroPt = {0, 0};
   WInfoPtr       infoPtr;
   TPrInfo        prInfo;

   infoPtr = (WInfoPtr)refCon;
   prInfo = (**(infoPtr->printRecord)).prInfo;

   newHeader.srcRect = prInfo.rPage;
   newHeader.hRes = FixRatio(prInfo.iHRes, 1);
   newHeader.vRes = FixRatio(prInfo.iVRes, 1);
   newHeader.version = -2;
   newHeader.reserved1 = 0;
   newHeader.reserved2 = 0L;

   err = SMPNewPage(&newHeader);
   if (err!=noErr)
      DoError(err);
   DrawAllShapes(infoPtr, zeroPt);
}
```

The SMPImage call takes care of including these image blocks for a letter. This routine is given the letter window and a draw-image routine, as well as a generic data pointer as input. The draw-image routine for CollaboDraw is called DrawImageProc; it accepts the window info block in the data pointer field. This callback first sets up the resolution and size of the page by extracting this information from the print record for the window. Next, SMPNewPage is called to set up the port into which the shapes will be imaged. Finally, the shapes are drawn into the page with the CollaboDraw routine DrawAllShapes, adding the final content blocks to the letter.

REPLYING TO OR FORWARDING A LETTER

Once a letter has been opened within CollaboDraw, several options are available. If additional correspondence is necessary, the letter can be replied to or forwarded. The mailer can also be removed, which turns the letter back into a document. This operation is very similar to closing a letter and is described in the next section. The following code is used to reply to a letter:

```
replyWindow = MakeWindow(kDrawMailerWindow, &newWindRect, newTitle,
      false);
err = SMPMailerReply(window, replyWindow, replyToAll, topLeft, true, true,
      kDefaultIdentity, nil, 0L);
```

```
if (err!=noErr)
   DoError(err);
ShowWindow(replyWindow);
```

The first step in replying to a letter is to make a new window in which the reply letter will be created. The CollaboDraw routine MakeWindow is called to create a new window of the mailer class. Once this window has been created, SMPMailerReply can be called, which takes the existing letter window, the new letter window, and several other parameters as input. As a result of the call, the reply letter is created and automatically addressed to the originator of the original message.

The mail forwarding process does not involve the creation of a new letter window. Instead, another mailer is added to the existing letter, and the mailers can be viewed by clicking a dog-ear in the corner of the mailer window pane. The code to forward a letter is as follows:

```
void CommForward(WindowPtr window)
{
   WInfoPtr infoPtr;
   char     hState;
   OSErr    err;

   infoPtr = BeginWindowAccess(window, &hState);
   HandleExpand(window, infoPtr);   // Expand window before doing forward.

   err = SMPMailerForward(window, kDefaultIdentity);
   if (err!=noErr)
      DoError(err);

   infoPtr->saved = false;
   DMailerActivateWindow(window, infoPtr, nil);
   EndWindowAccess(window, hState);
}
```

To forward a letter, the mailer in the window is first expanded. This allows the new mailer to be fully visible when it's created. The CollaboDraw routine HandleExpand calls SMPExpandOrContract to expand the mailer. Next, SMPMailerForward actually adds the mailer to the letter. Once this is done, the state of the document is changed to indicate that the letter is now an outgoing letter instead of a received letter. Finally, the activate-event method is called on the window to readjust the menu items that relate to sending mail.

CLOSING A LETTER

When it's time to close a letter window, there's a short process that must be adhered to. First, the optional close options dialog box can be displayed. This dialog gives the

user the option of deleting the letter or tagging it before it's closed. CollaboDraw uses the following code to display this dialog:

```
if (gPreferences.closeOptionsDialog) {
   SetCursor(&qd.arrow);
   err = SMPCloseOptionsDialog(window, &gPreferences.closeOptions);
   if (err!=noErr)
      returnValue = false;
}
```

Since the dialog box is optional, CollaboDraw has a preference variable that tracks whether the dialog should be displayed. If it should be displayed, this is done by calling SMPCloseOptionsDialog with the letter window and the close options to use when closing the letter. Note that the close options are also stored in the preferences, to allow the dialog to default to the close options last used.

The next step in the close process is to make sure that there are no open enclosures and that there are no Finder copies in progress that would prevent the closure of the letter. The following code excerpt checks for this:

```
err = SMPPrepareToClose(window);
if (err==kSMPHasOpenAttachments) {
   SetCursor(&qd.arrow);
   StopAlert(kHasOpenAttachID, nil);
   returnValue = false;
}
else if (err==kSMPCopyInProgress) {
   SetCursor(&qd.arrow);
   StopAlert(kCopyInProgress, nil);
   returnValue = false;
}
```

SMPPrepareToClose returns kSMPHasOpenAttachments if there are open enclosures and kSMPCopyInProgress if the user is in the process of copying a document to or from the enclosures list. In response, CollaboDraw presents an alert to the user and will not allow the letter to be closed.

The final steps in closing the letter are to remove the mailer from the window and close the window. Here's the code to do this removal:

```
err = SMPDisposeMailer(window, closeOptions);
if (err!=noErr)
   DoError(err);
return DrawDestroyWindow(window, infoPtr, data);
```

The PowerTalk routine SMPDisposeMailer removes the mailer pane from the window passed as input and releases all memory associated with the letter window. Once this is done, CollaboDraw calls the draw-window method for closing a window, which takes care of disposing of the rest of the window and document structures.

ADDING DIGITAL SIGNATURE PACKAGE SUPPORT

Digital signature services can also be incorporated into applications, providing a level of security not previously possible with personal computers. CollaboDraw allows signing and verifying within documents at a shape level. Individual shapes can be selected and signed, and the signatures are carried around with the shapes when the documents are saved or sent to other users. In addition to this shape-level digital signature support, the Standard Mail Package provides support for signing letters as a whole. By supporting the mailer, we automatically get this letter-based digital signature functionality.

SIGNATURE STORAGE FOR DOCUMENTS

Since digital signatures are quite large in size (they can be several kilobytes each), I elected not to store the signatures in memory with the document shapes. Instead, I store the signatures in the resource fork of each document file. The signature storage strategy is not covered in depth in the code below, but you can refer to the digital signature code within CollaboDraw to see how it's done.

SIGNING A SHAPE

To sign a shape or set of shapes within CollaboDraw, the user must select the shapes and choose the Sign Selected Shapes menu item. In response to this, the following code is called:

```
void CommSign(WindowPtr window)
{
   WInfoPtr       infoPtr;
   char           hState;
   ShapeListPtr   shapeList;
   SIGContextPtr  sigContext;
   Size           sigSize;
   OSErr          err;

   if (!IsAppWindow(window))
      return;

   err = SIGNewContext(&sigContext);
   if (err==noErr) {
      infoPtr = BeginWindowAccess(window, &hState);
      err = SIGSignPrepare(sigContext, nil, nil, &sigSize);
```


This article hasn't covered a few other PowerTalk features that CollaboDraw takes advantage of. Among these are printing, tagging letters, and opening the next letter from the mailbox. Refer to the sample code included on this issue's CD for implementation details of these features.•

```
      for (shapeList=infoPtr->data; (err==noErr) && (shapeList!=nil);
            shapeList=shapeList->next) {
         if (shapeList->selected) {
            err = SignShape(infoPtr, sigContext, shapeList, sigSize);
            InvalShapeArea(window, infoPtr, shapeList); // Redraw shape.
         }
      }

      SIGDisposeContext(sigContext);
      DSIGSetupSignMenu(window, infoPtr);
      EndWindowAccess(window, hState);
   }

   if (err!=noErr)
      DoError(err);
}
```

The first important call in the above code is SIGNewContext. This routine creates a digital signature context, which is required for each signing or verification session. Creating a context allocates the resources needed to perform signing and verification of objects.

Next, SIGSignPrepare is called. This routine prompts the user to enter a signer identification code, allowing the signer to be applied to the selected objects.

Now that the signature has been set up, each shape can be signed individually with a call to the CollaboDraw routine SignShape. Once each shape has been signed, SIGDisposeContext can be called to end the signing session.

The SignShape routine carries out the task of producing and storing a signature for each shape to be signed, as follows:

```
OSErr SignShape(WInfoPtr infoPtr, SIGContextPtr sigContext,
               ShapeListPtr theShape, Size sigSize)
{
   OSErr          err;
   Handle         signature;
   short          resID;
   short          saveResFile;
   DigSigListPtr  theSig;

   // Allocate storage for the signature.
   signature = NewHandleChk(sigSize);
   if (MemError()!=noErr)
      return MemError();
```

```
   // Process the data for the signature.
   err = ProcessShapeData(sigContext, theShape);
   if (err!=noErr) {
      DisposHandleChk(signature);
      return err;
   }

   // Create the signature.
   HLock(signature);
   err = SIGSign(sigContext, (SIGSignaturePtr)*signature, nil);
   HUnlock(signature);
   if (err!=noErr)
      return err;

   // Add the signature to the shape.
   saveResFile = CurResFile();
   UseResFile(gDSTempRefNum);
   resID = Unique1ID(kSignatureResType);
   AddResource(signature, kSignatureResType, resID, "\p");

   . . .
}
```

Before a shape can be signed, memory must first be allocated to hold the signature for the shape. The size of the block required to hold the signature is returned by SIGSignPrepare, and this value is passed in as the sigSize parameter to the SignShape routine.

Once the signature storage has been allocated, all of the data to be signed in the shape must be handed to the Digital Signature Manager in a byte stream. This process is required to generate a unique number identifying the contents of the document. The CollaboDraw routine ProcessShapeData handles this data streaming. Within ProcessShapeData, the Digital Signature Manager routine SIGProcessData is called to stream the data.

```
err = SIGProcessData(sigContext, theShape, kShapeSignLength);
```

Once the unique number, also known as a *digest*, has been created, the signer is then applied to that number to create a signature with the call SIGSign. The signature is stored in the handle allocated at the start of the routine and is then added to the resource fork of the document file.

VERIFYING A SHAPE

Once a shape has been signed, it can later be verified from within CollaboDraw. The high-level CollaboDraw routine CommVerify is called in response to the Verify

Selected Shapes menu item. This routine is almost identical to the CommSign routine given earlier, so it isn't included here. It simply calls SIGNewContext and then repeatedly calls the CollaboDraw routine VerifyShape. Once each shape has been verified, CommVerify calls SIGDisposeContext.

The VerifyShape routine is analogous to the SignShape routine. Instead of adding a signature to a shape, this routine retrieves the signature for a shape, verifies the signature, and displays information about the signer.

```
signatureSize = SizeResource(sigHandle);
HLock(sigHandle);
err = SIGVerifyPrepare(sigContext, (SIGSignaturePtr)*sigHandle,
   signatureSize, nil);
if (err==noErr) {
   // Process the data for the signature.
   err = ProcessShapeData(sigContext, theShape);
   if (err==noErr) {
      err = SIGVerify(sigContext);
      if (err==noErr)
         err = SIGShowSigner(sigContext, nil);  // Show signer info.
   }
}
```

The section of VerifyShape shown above comes just after the signature is extracted from the resource fork of the document. Once the signature is in sigHandle, SIGVerifyPrepare is called. This routine prepares the Digital Signature Manager to receive data via the SIGProcessData call. Once this data has been streamed to create a digest, the digest is compared to the one stored in the signature with the SIGVerify routine. This routine will return noErr if the two digests match. When this occurs, a SIGShowSigner call will present a dialog box displaying information about the signer of the shape.

EXPLORING OTHER POWERTALK FEATURES

This discussion of the PowerTalk Standard Mail and Digital Signature packages doesn't even begin to touch on the many features available to developers through PowerTalk. You can take advantage of InterProgram Messaging for store and forward application communication, use the Standard Catalog interfaces for picking items out of catalogs, write custom catalog templates, use PowerTalk authentication services, or build service access modules to interface to alternate message delivery or directory catalog services. By adding standard mail and digital signature support to CollaboDraw, we've enhanced the usefulness of our simple drawing program in many ways. When combined with other applications that support PowerTalk collaborative services, communication and productivity within a workgroup can be taken to new levels.

THANKS TO OUR TECHNICAL REVIEWERS
Godfrey DiGiorgi, John Evans, Steve Fisher, Martin Minow•

THE VETERAN NEOPHYTE

ABRACADABRA

DAVE JOHNSON

I've just returned from a really long vacation. For six weeks I didn't touch a single computer. (Well, that's not strictly true; I did stroke many a touch-screen on information kiosks or ticket machines, but you get the idea.) The first time after my return that I grabbed the mouse of a live Macintosh there was a brief instant — just a single, sharp, fleeting moment — when I felt the magic again.

Can you remember the first time you got to play with a working Macintosh? Were you amazed — I mean really astounded — as I was? Did you: Peek under the mouse to see what was there? Click and drag all over the place just to watch things happen? Drag a file into a folder and then immediately open the folder to see if the file was really there? Create a nest of new folders deep enough to get bored, just to see if it would work? Try every combination of bold, outline, shadow, italic, and underline?

I'm betting that the fundamental reason you're interested in programming the Macintosh is because of that magic. I know this isn't true for everyone out there (some of you — gasp — probably do it for the money!), but I suspect it's true for most of you, or at least it was when you started. Maybe you wanted to make a little of that magic yourself. Maybe you just wanted to peek behind the curtain to see how it was done. Or maybe you wanted (as I did) to find out where the magic came from, to hunt down its source. One of the problems with that kind of techno-magic, though, is that the more you learn about it and the more you use it, the more it fades away.

So here's the next question: When was the *last* time you felt the magic? If you're like me, the magic of the Macintosh interface has been completely subsumed by everyday familiarity. It's become a part of everyday life, like matches, or light bulbs, or TV. I'm sure that when matches were still new, striking one and making fire was an amazing thing. I'll bet people used up whole boxes of matches, striking them one by one, just to see it happen. But matches are no longer special; their magic has become cheap and commonplace and has therefore ceased to be magic at all. People don't light matches for the thrill anymore (pyromaniacs excepted); they use them to light something else — matches have become a means, not an end. Similarly, we don't marvel anymore at the fact that just by flipping a switch we can make an entire room as bright as day, banishing forever the night; we think instead about what we want to do tonight. We don't marvel anymore that moving pictures and sounds can be plucked out of the air (or out of a cable, these days) and made to show up on a box in our homes; we think instead about what's on.

This is probably a necessary and inevitable step in a culture's acceptance and assimilation of a new technology: people stop marveling at the fact that they have a new ability, and begin simply to *use* that ability. That period when new technologies still feel like magic is also the period when a culture is adjusting itself to the technology and being transformed by it. By the time a new technology has been fully integrated into society, it's taken for granted, the magic exhausted and the transformation complete.

So how does this apply to computers? Is the magic from computers all used up? Have they been fully assimilated by human society and finished their transformational work? Are they now taken for granted and just a part of the background noise of modern life? In the words of my mom when I asked her (at age 11) if I could get a tattoo on my chest: Hell, no.

DAVE JOHNSON wants to know: is he the only one who does watch-cursor push-ups during time-consuming Macintosh operations? First you find a horizontal black line (they're everywhere: window frames, folder icons, buttons, even the progress bar itself), then you put the watch cursor just above it, so that the bottom edge of the watchband overlaps the horizontal line by one pixel. Now carefully move the cursor up and down by one pixel, and there you have it — watch-cursor push-ups! You can do pull-ups too! Amaze your friends!•

Galileo's finger is preserved in a bottle, just like a holy relic, in a science museum in Florence, Italy. I saw it myself. Really.•

Particular *manifestations* of computers have become a part of daily life for many people: cash machines, video games, bar code readers at markets, and so on. These are computers, but they're masked — the true nature of the machine is obscured by a task-specific facade. Even the relatively small number of people who use "real" computers in their everyday lives use them for only a few tasks (word processing, graphics editing, number crunching, and game playing are common — somehow recipe filing never caught on). So they're really just using two or three specific, task-oriented applications. And yes, these particular uses of computers have become mundane to those who use them: writers use word processors without blinking, accountants use spreadsheets without a hint of awe.

But I'm not sure whether computers *as computers* can ever be fully integrated into society. They're too slippery, too prolific, too, well, *protean*. (Protean: able to take on new forms easily, after Proteus, a sea god in Greek mythology who could change his shape at will.) Just when we get used to them in one guise, they blur and shift and suddenly they're something else, something new, something magical all over again.

And that's where programmers come in. We're the ones who get to cause that shift. We're the ones who get to craft new faces for the machine, like mad, happy mask makers. We're the ones that get to *make the magic*. We get closer than anyone else to tasting the real flavor of computers — their malleability and chameleon-like talent for taking on new forms — but it's still only a taste, and the price is outrageous. Making magic turns out to be nothing but hard, grungy work. Being a wizard looks great from the outside, but there's a downside most people don't see: to create the magic, you need to spend inordinately huge amounts of time doing completely unmagical things, and even worse, you have to give up experiencing the magic for yourself. It's like sleight of hand: it looks like magic to the audience, but to the conjurer it's not magic at all. Learning that kind of magic means spending countless hours alone in front of a mirror, practicing the same moves over and over and over until they're automatic and can be made without even thinking. By that time any residual magic has been completely wrung out of it.

Like brain researchers who set off to find the source of human consciousness and end up studying the function of some enzyme in sea slugs, programmers often set off to find the source of the magic and end up writing device drivers. There's a valuable lesson there, one that took me years to learn: the magic isn't part of the machine at all. You can follow the computer's workings right down to the bottom, and what you find is a boringly predictable mechanism as devoid of magic as a meat grinder. It's like trying to find musical beauty by closely examining a CD: all you can find is a series of rough pits in a reflective surface, and there's no indication whatever that those pits could contain something sublime.

So where does the magic come from? The answer's obvious, once you stop to think about it: it comes from people. It turns out that computers don't possess any magic of their own, they're just very, very good containers for human magic. The computer is simply a shell, albeit one that's infinitely malleable. The people who shape the shell, who tell the computer what to be, are the real source of the magic. I guess I should've known.

RECOMMENDED READING

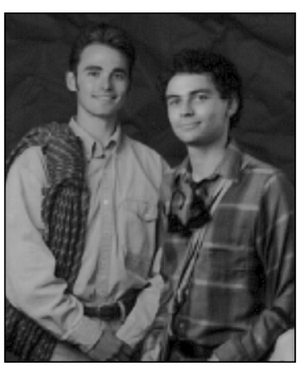
- *Man Meets Dog* by Konrad Lorenz (Penguin Books, 1964).
- *The Phantom Tollbooth* by Norton Juster (Random House, 1964).
- *Let It Rot!* by Stu Campbell (Storey Publishing, 1975).

Thanks to Jeff Barbose, Michael Greenspon, Bill Guschwan, Mark ("The Red") Harlan, Bo3b Johnson, Lisa Jongewaard, and Ned van Alstyne for their always enlightening review comments.•

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.•

DRAG AND DROP FROM THE FINDER

Some users navigate Standard File dialogs with no problem, but many others find them tedious or even confusing. Users want to find and organize their files without having to learn the intricacies of the hierarchical file system and Standard File dialogs. With applications that use the Drag Manager, users will be able to work with files the way they expect: by dragging files from the Finder into an application window. This article and the accompanying sample program show how easily your application can provide this valuable feature.



DAVE EVANS AND GREG ROBBINS

The Drag Manager is so new you won't find it in *Inside Macintosh* yet, but if your application works with files, you'll want to learn more about what it can do for you. This new Macintosh Toolbox manager lets users drag and drop data (such as text and pictures) between windows in an application and between different applications. It also allows users to drag document icons to and from the Finder.

Rather than describing the Drag Manager in depth, this article and its sample application focus on using the Drag Manager to drag picture files (files of type 'PICT') from the Finder into an application. The techniques used by the sample application can easily be generalized to cover other cases.

The Drag Manager is currently packaged as a system software extension that you can license to include with your products. It requires System 7, and to take advantage of the Finder dragging described in this article you need Finder version 7.1.1 or later. You can order the full Macintosh Drag and Drop Developer's Kit from APDA. The Drag Manager will also be included in future system software releases.

Along with the sample application, called SimpleDrag, this issue's CD contains the programmer's guide for the Drag Manager as well as the *Drag and Drop Human Interface Guidelines*. After you've read this article and looked at the SimpleDrag code, you should read these two documents to get a deeper understanding of the Drag Manager. SimpleDrag not only allows picture files to be dragged from the Finder but

DAVE EVANS can often be found coding for the User Experience team of the AppleSoft OS Platform Group. Although some still think he moonlights on the set of the TV show "Beverly Hills 90210," Dave actually finds entertainment by throwing himself off cliffs and cornices, plane struts and buildings (the last much to Apple Security's chagrin). Dave does admit, though, to being deathly afraid of bungee jumps!•

GREG ROBBINS has been insisting for three years that he doesn't work for Apple. But he has worked as a consultant to the Developer Technical Support and Macintosh System Software groups since 1991, having given up an earlier passion for neural networks to hack the Mac. Greg spends his off hours in the mountains of California, looking for people even more lost than he is.•

also lets PICT data be dragged from one application window into another; for the full story on this, look at the code and documentation on the CD.

THE INTERFACE: IT'S NOT SUCH A DRAG

Dragging is a skill that every Macintosh user has mastered. It provides a quick, simple alternative to commands as a way of performing common operations such as moving or deleting files. This use of dragging gives users a sense of control because they can manipulate objects directly with excellent visual feedback. And it's faster and more intuitive than commands because it's not hidden in a menu.

Since all Macintosh users use dragging to arrange and manipulate files in the Finder, it only makes sense that they should be able to drag files from the Finder into an application window. But until now, the only way to select and specify files within an application has been with Standard File dialogs. Now you can use the Drag Manager to provide an alternate, more intuitive way to work with files: the user can open a file in your application simply by dragging the file's icon into an application window.

FIRST, A FEW TERMS

Before we look at the sample code, we need to clarify a few terms that the Drag manager introduces: *drag items*, *drag flavors*, *drag handlers*, *senders*, and *receivers*.

The objects that a user drags are called *drag items*. For example, a user who selects and drags three files is dragging three different drag items.

Drag flavors describe the kind of data that a drag item contains. When a user drags an item to an application window, the receiving application must determine whether it can accept the data in the drag item. Each item can have more than one flavor, because data can usually be described in more than one format or data type. For example, you can describe text data as ASCII data, styled text data, or RTF interchange format; if a program can't accept the more elaborate RTF format, it may be able to use the plain ASCII text. The Drag Manager uses a four-character ResType to identify a flavor. In our sample application, we use only two drag flavors: one that identifies files dragged from the Finder, and another that identifies PICT data dragged from an application window.

The Drag Manager uses an application's *drag handlers* to provide dragging feedback and to complete a drag. There are two types of drag handlers: *tracking handlers* and *receive handlers*. A tracking handler is called while an item is being dragged over an application's windows; a receive handler is called when the user releases the mouse button to drop the item in a window. Each window has a tracking handler and receive handler installed for it, though several windows may use the same handler. When you initialize your application or open a new window, you call the Drag Manager to install your drag handler callback routines.

Because the Drag Manager provides interapplication drag and drop services, it's important to know where the drag starts and where it ends. The application in which the drag starts is called the *sender*. Any application that the item is dragged over is a potential *receiver* of the drag; the application it's dropped into is the actual receiver. The sender and receiver might be the same application — but with interapplication dragging, another application could be the receiver of the drag.

NOW, ON TO THE CODE

With the lingo out of the way, let's look at our SimpleDrag application. This application displays pictures in its windows. One way the user specifies a picture file to be displayed is by choosing the application's Open command and then selecting a file from the Standard File dialog. But since the application uses the Drag Manager, the user can also drag a picture file from the Finder into a SimpleDrag window. PICT data displayed in a SimpleDrag window can even be dragged into another SimpleDrag window.

First let's consider the code for the Open command case. When the user chooses the Open command, SimpleDrag calls the Standard File Package to present a dialog that lists the picture files. Once the user has selected a file, SimpleDrag calls its SetWindowPictureFromFile routine to read the file and display it.

To support dragging files from the Finder into the application, SimpleDrag installs two drag handlers for each new window. While the user drags a PICT drag item over a SimpleDrag window, the tracking handler provides visual feedback. If the user drops the item in a SimpleDrag window, the Drag Manager calls the receive handler to read and display the PICT information, which may be not only a picture file but also PICT data dragged from another window; the receive handler calls its SetWindowPictureFromFile routine if the drag item is a picture file (just as when the user chooses Open from the File menu).

The following routine installs the tracking and receive handlers:

```
OSErr InstallDragHandlers(WindowPtr theWindow)
{
   OSErr retCode;

   retCode = InstallTrackingHandler(MyTrackingHandler, theWindow, nil);
   if (retCode == noErr) {
      retCode = InstallReceiveHandler(MyReceiveHandler, theWindow, nil);
      if (retCode != noErr)
         (void) RemoveTrackingHandler(MyTrackingHandler, theWindow);
   }
   return retCode;
}
```


Before you call any Drag Manager routines, make sure that the Drag Manager is available by calling Gestalt with the selector gestaltDragMgrAttr and checking the gestaltDragMgrPresent bit of the response.•

That's all you need to do to set up tracking and receive handlers for the given window. You can also install a default handler, to be used for any window that you don't explicitly install a handler for, by passing nil as the window pointer to the install routine.

TRACKING THE DRAG

Now let's see what happens while the user drags an item around. Our main objective is to indicate, with visual feedback, where it's OK to drop the item. SimpleDrag provides the standard feedback highlighting for its type of windows and data — a thin frame highlight within the content region of the window. This highlight signals the user that the item can be dropped there.

While the user drags an item (or items) over one of the application's windows, the mouse movement determines what messages the tracking handler receives, as follows:

- The tracking handler receives an EnterHandler message the first time it's called (that is, the first time the drag enters a window that uses that handler). You can allocate memory or, as in our application, check whether you can receive the drag.
- The handler receives an EnterWindow message when the drag enters a window. This message is distinct from EnterHandler because you may be using the same handler for more than one window, in which case there might be many EnterWindow messages between an EnterHandler/LeaveHandler pair.
- While the user drags within a window, the handler receives multiple InWindow messages.
- When the drag leaves the window, the handler receives a LeaveWindow message.
- When the user drags to a window that uses a different tracking handler, the handler receives a LeaveHandler message.

The tracking handler for SimpleDrag is as follows:

```
pascal OSErr MyTrackingHandler(DragTrackingMessage theMessage,
   WindowPtr theWindow, void *handlerRefCon, DragReference theDrag)
{
#pragma unused (handlerRefCon)

   RgnHandle   tempRgn;
   Boolean     mouseInContentFlag;
   OSErr       retCode;

   retCode = noErr;
```

```
switch (theMessage) {
   case dragTrackingEnterHandler:
      // Determine whether the drag item is acceptable and store
      // that flag in the globals, plus reset the highlighted global
      // flag.
      gDragHandlerGlobals.acceptableDragFlag =
         DragItemsAreAcceptable(theDrag);
      gDragHandlerGlobals.windowIsHilightedFlag = false;
      break;

   case dragTrackingEnterWindow:
   case dragTrackingInWindow:
   case dragTrackingLeaveWindow:
      // Highlighting of the window during a drag is done here. Do it
      // only if we can accept this item and we're not in the source
      // window.
      if (gDragHandlerGlobals.acceptableDragFlag &&
            DragIsNotInSourceWindow(theDrag)) {
         if (theMessage == dragTrackingLeaveWindow)
            mouseInContentFlag = false;
         else
            mouseInContentFlag = MouseIsInContentRgn(theDrag,
                                                     theWindow);
         if (mouseInContentFlag &&
               !gDragHandlerGlobals.windowIsHilightedFlag) {
            ClipRect(&theWindow->portRect);
            tempRgn = NewRgn();
            RectRgn(tempRgn, &theWindow->portRect);
            if (ShowDragHilite(theDrag, tempRgn, true) == noErr)
               gDragHandlerGlobals.windowIsHilightedFlag = true;
            DisposeRgn(tempRgn);
         }
         else if (!mouseInContentFlag &&
               gDragHandlerGlobals.windowIsHilightedFlag) {
            ClipRect(&theWindow->portRect);
            if (HideDragHilite(theDrag) == noErr)
               gDragHandlerGlobals.windowIsHilightedFlag = false;
         }
      }
      break;

   case dragTrackingLeaveHandler:
      // Do nothing for the LeaveHandler message.
      break;
```

```
      default:
         // Let the Drag Manager know we didn't recognize the message.
         retCode = paramErr;
   }
   return retCode;
}
```

To give the user visual feedback, the tracking handler uses the Drag Manager's ShowDragHilite routine. This routine takes a region to be highlighted and draws an inset or outset frame of that region. Here we use it to highlight inside the content region of the window, but you can also use it to highlight panes within a window or any arbitrary region that accepts a drag. We later call HideDragHilite when the drag leaves the content region of our window.

As you can see in the above code, there are several conditions to check for before calling the highlight routines. The DragItemsAreAcceptable routine, which the tracking handler calls when it gets an EnterHandler message, checks that only one item is being dragged (a limitation of our simple example) and that the drag item is PICT data or a picture file.

```
Boolean DragItemsAreAcceptable(DragReference theDrag)
{
   OSErr          retCode;
   unsigned short totalItems;
   ItemReference  itemRef;
   Boolean        acceptableFlag;
   HFSFlavor      currHFSFlavor;
   Size           flavorDataSize;
   FlavorFlags    currFlavorFlags;

   acceptableFlag = false;

   // This application can only accept the drag of a single item.
   retCode = CountDragItems(theDrag, &totalItems);
   if (retCode == noErr && totalItems == 1) {
      retCode = GetDragItemReferenceNumber(theDrag, 1, &itemRef);
      if (retCode == noErr) {
         // Use GetFlavorFlags to see if the drag item is PICT data.
         retCode = GetFlavorFlags(theDrag, itemRef, 'PICT',
                                  &currFlavorFlags);
         if (retCode == noErr)
            acceptableFlag = true;
         else {
            // Check if the item is a file spec for a picture file.
            flavorDataSize = sizeof(HFSFlavor);
```


The EnterWindow message is sent when the drag enters the structure region of a window, not the content region. The *Drag and Drop Human Interface Guidelines* specify that the title bar of a window, which is outside the content region, should not be able to receive drags. So upon receiving an EnterWindow message, the tracking handler needs to check the mouse location before calling ShowDragHilite.•

```
            retCode = GetFlavorData(theDrag, itemRef, flavorTypeHFS,
                                    &currHFSFlavor, &flavorDataSize, 0);
            if (retCode == noErr && currHFSFlavor.fileType == 'PICT')
               acceptableFlag = true;
         }
      }
   }
   return acceptableFlag;
}
```

DragItemsAreAcceptable calls GetFlavorFlags with type 'PICT' to determine whether the drag item is PICT data. If it isn't PICT data, GetFlavorFlags returns cantGetFlavorErr; DragItemsAreAcceptable then checks to see if the drag item is a picture file, by calling GetFlavorData with flavorTypeHFS. This is a special flavor that identifies files dragged from the Finder into an application. Data of type HFSFlavor contains the file's Finder information and an FSSpec that you can use to open and read the file.

```
typedef struct HFSFlavor {
   OSType         fileType;         // file type
   OSType         fileCreator;      // file creator
   unsigned short fdFlags;          // Finder flags
   FSSpec         fileSpec;         // file system specification
};
typedef struct HFSFlavor HFSFlavor;
```

Another check made in the tracking handler is to ensure (with the routine MouseIsInContentRegion) that the drag isn't over the title bar or over controls in the application window. To implement drag and drop according to the guidelines, we accept drags only in the content region of a window. Also, since SimpleDrag doesn't support drag and drop within the same window, the tracking handler checks (with its DragIsNotInSourceWindow routine) to make sure that the user isn't dragging over the same window in which the drag originated.

RECEIVING THE DRAG

The receive handler is similar to the tracking handler, but it's called once, and we must ask for all the data we want. We also make sure that the drag stopped in the content region of the window and that the user isn't dragging back into the source window.

Below is the code for SimpleDrag's receive handler. In a receive handler, you first ask for the data type you prefer, whether picture, text, or some other type, and whether a file from the Finder or data dragged directly from another window. In SimpleDrag, we prefer to receive PICT data directly, so we look for it first. If the drag item isn't PICT data, we use the HFS flavor to look for files of type 'PICT'.

```
pascal OSErr MyReceiveHandler(WindowPtr theWindow, void *handlerRefCon,
      DragReference theDrag)
{
#pragma unused (handlerRefCon)

   ItemReference  itemRef;
   Size           dataSize;
   Handle         tempHandle;
   HFSFlavor      theHFSFlavor;
   Boolean        dataObtainedFlag;
   OSErr          retCode;

   dataObtainedFlag = false;
   if (!DragItemsAreAcceptable(theDrag) ||
         !MouseIsInContentRgn(theDrag, theWindow) ||
         !DragIsNotInSourceWindow(theDrag))
      return dragNotAcceptedErr;

   // There is only one item, so get its reference number.
   retCode = GetDragItemReferenceNumber(theDrag, 1, &itemRef);
   if (retCode != noErr)
      return retCode;

   // PICT data is preferred, so get it if it's available.
   retCode = GetFlavorDataSize(theDrag, itemRef, 'PICT', &dataSize);
   if (retCode == noErr) {
      tempHandle = TempNewHandle(dataSize, &retCode);
      if (tempHandle == nil)
         tempHandle = NewHandle(dataSize);
      if (tempHandle != nil) {
         HLock(tempHandle);
         retCode = GetFlavorData(theDrag, itemRef, 'PICT', *tempHandle,
                                                  &dataSize, 0);
         if (retCode == noErr) {
            retCode = SetWindowPicture(theWindow, (PicHandle) tempHandle);
            if (retCode == noErr)
               dataObtainedFlag = true;
         }
         DisposeHandle(tempHandle);
      }
   }

   if (!dataObtainedFlag) {
      // Couldn't get PICT data so try to get HFS-flavor data.
      dataSize = sizeof(HFSFlavor);
```

```
      retCode = GetFlavorData(theDrag, itemRef, flavorTypeHFS,
                                    &theHFSFlavor, &dataSize, 0);
      if (retCode == noErr && theHFSFlavor.fileType == 'PICT') {
         retCode = SetWindowPictureFromFile(&theHFSFlavor.fileSpec,
                                                              theWindow);
      }
   }

   if (retCode != noErr)
      (void) ReportErrorInWindow(nil,
                        "\pCannot display received picture. ", retCode);
   return retCode;
}
```

If there's an error, this receive handler just displays a simple string. For commercial products, you would never code strings inline as shown, for localization reasons.

GOTCHAS

Here we'll describe a couple of precautions you should take that will make your life easier when you use the Drag Manager.

FINDER GOTCHAS

The Drag Manager works for documents and other standard files, but what about folders and hard drive icons? The Finder uses the same HFS flavor to describe these items. If the user drags them to your application, you'll see the FSSpec for the folder's directory or the disk's root directory. The file type and creator information isn't relevant to the file system, but it's useful for identifying the items being dragged. For both folders and disk icons, the creator is set to 'MACS' to show that the system software created them. For folders, the file type is 'fold', and for disk icons the file type is 'disk'. In both cases the Finder flags for the folder or disk are set appropriately. Remember that these file types serve only to quickly identify the items being dragged and don't reflect what's in the catalog information of any volumes.

Some software that extends the functionality of the Finder, such as QuickDraw GX and PowerTalk (the client server software based on the Apple Open Collaboration Environment), adds new Finder icons such as desktop printers, letters, and mailboxes. These items don't actually represent the state of the file system, but they can be dragged like any Finder icon. This is a valuable and consistent metaphor for the Finder interface, but it creates an inconsistency for your receive handler when receiving drags from the Finder. Since these icons can't be described as FSSpecs, don't expect to receive HFS flavors for them.

Just for completeness, you should know that the Users & Groups control panel also uses the Drag Manager. The drag flavors that identify those icons make sense only to

For more on letters and mailboxes, see the article "Building PowerTalk-Savvy Applications" in this issue of *develop.*•

the Finder, and don't have relevant information you could extract. The same is true for contents of Finder suitcase files like the System file. Finder icons for sounds, keyboard layouts, and fonts that are in suitcases are representations of resources in the suitcase file, so they don't have HFS flavors to describe them. Note, however, that sound and font *file*s, which are not part of suitcases, use HFS flavors just like any other file.

WAITNEXTEVENT

Another precaution applies if, in drag handlers, you call WaitNextEvent, EventAvail, GetNextEvent, or any other routine that would normally cause a process switch or a background application to receive WaitNextEvent time. In these cases, don't expect other applications to receive any background time, because the Drag Manager disables process switching during a drag. Because process switching is disabled, you should be careful when interacting with the user in your receive handler. You may not be the frontmost process, and opening a dialog may hang the Macintosh.

DRAGGING AWAY

The Drag Manager makes it easy to add drag and drop functionality to your application. It gives users a familiar and intuitive way to manipulate files and data. This article and the sample application emphasize how to implement dragging files from the Finder into your application windows, but you can do much more than that with the Drag Manager. So take a look at the documentation and guidelines on the CD and give it a go; your users will think it's anything but a drag!

THANKS TO OUR TECHNICAL REVIEWERS
Steve Fisher, Rob Johnston, Jim Mensch, Andy Nicholas•

PRINT HINTS

LASERWRITER 8 FOR FUN AND PROFIT

MATT DEATHERAGE

In May of this year, Apple and Adobe Systems released version 8.0 of the LaserWriter driver — the biggest change to PostScript® printing on the Macintosh in eight years. This new driver was rewritten from the ground up by engineers at Adobe, working closely with Apple engineers who provided source code, guidance, and comprehensive testing. The resulting driver unleashes the power of PostScript Level 2 printing as much as possible under the limitations of the pre–QuickDraw GX printing architecture.

Since the driver has been in general release for quite some time, we won't go through feature lists or repeat things you've seen printed elsewhere. Instead we'll focus on how your application can compatibly take advantage of the new driver's power. We'll also note some programming practices that cause problems with LaserWriter 8 and that will continue to cause grief in the future with QuickDraw GX.

WHERE TO FIND INFORMATION

The best news for programmers about this driver is that there's more information available for this driver than for any printer driver before it. It follows the standard Printing Manager API documented in *Inside Macintosh* Volumes II and V, which you should stop reading immediately when you can grab *Inside Macintosh: Imaging With QuickDraw*. The new book has vastly superior organization and information about printing, including what's covered in the older Macintosh Technical Notes, Q&As, and other sources of information Apple has refined through the years. Even if you don't yet have the new book, everything in the older volumes and in the Technical Notes is still valid, with few exceptions.

In addition, this issue's CD contains a 25-page document called "Developer Information," located in the LaserWriter 8 folder, that talks about differences between the old and new drivers. Since this driver was in open beta test from October 1992 through May 1993 as "PSWriter," the changes shouldn't surprise too many people. The "Developer Information" document has been available since October 1992 as well, so all kinds of great information have been at your fingertips for quite some time.

THE GROUND RULE

The new driver has several exciting new API calls to assist with PostScript printing and translation. However, before getting into them, we must state The Ground Rule: *While you're encouraged to use new LaserWriter 8 features where appropriate, your application should* not *depend on them. Your handling of printing or key services should not fail if the features aren't available.*

Some users may not have upgraded to LaserWriter 8, and some may be running specialized versions of the older driver and don't want to give it up. More important, however, is that the new API features provided through PrGeneral are *not* present in QuickDraw GX. In its initial release, QuickDraw GX makes it easy for PostScript printer manufacturers to create their own drivers supporting all their printer-specific features; under the old architecture this was about as easy as removing your eyeballs from their sockets with your toes. This means there's a chance that a customer will have software *beyond* LaserWriter 8 that doesn't support the new calls documented on the CD, so use them only if they're present.

For example, if you have an application that uses PostScript printer description (PPD) files to obtain information about a target printer, you can use the new driver's PSPrimaryPPDOp selector to PrGeneral to

---

MATT DEATHERAGE (AppleLink DEATHERAGE1) has been doing technical support for over five years. In addition to serving as the technical lead for the LaserWriter 8 driver in Apple's Developer Technical Support group, he's worked on several Apple printer projects when not focusing on fonts or typography or all the other fun imaging stuff there's never enough time for. He remains true to his background by running the Apple II Programmers and Developers roundtable on GEnie, but you can find him on-line in lots of other places, usually just where you're hoping he won't be.•

While we refer to the driver here as "LaserWriter 8," Adobe distributes the same driver as "PSPrinter." Adobe also licenses the driver to printer manufacturers who license PostScript language software from Adobe, and those printer makers may call the driver by different names as well. Apple's initial release of the driver is named "LaserWriter 8.0," but later releases (which may happen by the time this is published) will be named "LaserWriter 8." No matter what the driver is called or how the icons appear, the internals are the same and all the information here applies.•

obtain an FSSpec referencing the PPD file the user chose for the current printer in the Chooser. However, if the driver isn't available, the new PrGeneral calls return the error opNotImpl (opcode not implemented) or resNotFound (the current driver doesn't support PrGeneral). In this case, you should ask the user to locate the PPD file just as you did before LaserWriter 8 was available. You should *not* display an alert saying "You have the wrong printer driver chosen; go fix it."

Another example is in the powerful new functions that the driver provides for converting QuickDraw pictures to encapsulated PostScript (EPS) format. If using these functions allows you to export EPS data in a cross-platform document for higher fidelity on another system, that's great. If the driver isn't available, though, you need to create your own EPS data or provide some alternate representation of the data (perhaps a bitmap or a TIFF image).

Remember, these functions are there to use if they're available, but you can't require them. It's not fair to remove features from your application based on the printer driver version. Try to use them if you need to, but be prepared for when they're not present.

Sample code that helps explain the new features is also on the CD, in the same folder as the new driver.

NEW THINGS YOU CAN DO

With that caveat on the table, let's discuss how some of the new features might affect your program. There isn't room here to cover all the new APIs or their uses, so be sure to check out the documentation and sample code on the CD for the comprehensive scoop.

PostScript printer description files. In addition to using PPD files in applications, some people will need to create PPD files for custom in-house purposes, or because they're developing PostScript-compatible hardware. There's a trick that's undocumented (and unguaranteed, so don't write code that depends on this!) which lets the driver help you debug your PPD files: Normally, when the driver parses a PPD file (during Chooser selection), it puts up a simple error dialog saying "This PPD is invalid" if it finds a problem with the file. However, the driver has a resource of type 'PLRT' with ID 1 that contains a single byte, normally 0. If you change this byte to $01, the driver will display an alert during PPD parsing saying exactly what it didn't like and on what line it didn't like it. Since PPD files can get fairly complicated in short order, this feature can be extremely helpful.

This preference is stored in a PLRT resource only in versions 8.0 and 8.0.1 (shipping with Adobe Acrobat). In later releases, it will be combined with some other useful preferences in a PRFS resource. There's a resource editor TMPL template describing the bits in the PRFS resource. One of these bits tells the driver to write all Printer Access Protocol (PAP) transactions to a PAPToDisk file so that you can debug your printing code's output. This lets you see what the driver generates during normal printing (not what it generates in the special case of creating a PostScript file) and is quite handy. Have fun with the preferences, but remember not to ship code that relies on them — don't modify them from your code.

Dealing with EPS files. The older LaserWriter driver had been creating PostScript files as a debugging aid for several years, and in version 7.0 this feature was finally added to the print job dialog so that users could visibly control it. In 8.0, the file creation mechanism has been further improved: it now provides user control over PostScript language level, font inclusion, and ASCII or binary protocol, and can create EPS files on request. Since so many applications understand EPS files, this feature gets a lot of exercise, and fosters some misunderstandings. The encapsulated PostScript file format version 3.0 is discussed in Appendix H of Adobe's *PostScript Language Reference Manual*, Second Edition (the "red book"). The red book clearly states that an EPS file describes an image of a single page. That's why printing a range of pages to an EPS file gives you an image of only the first page — you can't have an image of more than one page in an EPS file.

The main problem users are having with EPS files created by the driver, however, is with the EPS preview.

For more information on PrGeneral, see the article "Meet PrGeneral, the Trap That Makes the Most of the Printing Manager" in *develop* Issue 3.•

Information on PostScript printer description files and other items related to PostScript language development are available from the Adobe Systems Developers' Association, 1585 Charleston Road, P.O. Box 7900, Mountain View, CA, 94039-7900, telephone (415)961-4111.•

This preview is an optional PICT resource of ID 256 which, if present, contains a QuickDraw picture resembling what a PostScript interpreter would produce after executing the EPS code.

LaserWriter 8 does previews in three ways: it creates a picture that's one giant bitmap or pixel map ("standard preview"); it creates a picture containing all the drawing commands used to draw the page ("enhanced preview"); or it doesn't create a preview at all. The choice is the user's, or at least it's *supposed* to be. Some applications don't like the word "optional" very much and refuse to import EPS files that don't contain the "optional" preview. Don't make the mistake of considering the preview to be required.

Since the information for the preview picture comes from the drawing your application does into the printing grafPort, there's trouble if your application doesn't draw things as the driver expects. For example, some applications still examine the high byte of the wDev field in the print record and, if they find the value 3, assume they're talking to a PostScript printer. Such programs send all their data to be printed as custom PostScript code, not drawing anything into the printing grafPort. This makes for an extremely boring preview image — your code didn't draw anything, so there's nothing in there to display.

If you create your own PostScript code, *always* send dual QuickDraw and PostScript information when printing or exporting pictures. Never create PostScript code without making as faithful a QuickDraw representation as you can. Not only does this prevent EPS preview problems with LaserWriter 8, it prevents problems under QuickDraw GX, where users can redirect print files after you've finished your print loop but before they're imaged on a printer. If you sent only PostScript code and the user redirects the print job to a StyleWriter II because the PostScript printer is busy, the result will be a bunch of blank pages.

Some people have asked why EPS files created by the driver seem to be the size of a printed page when the real image is only a small part of the page. That happens when your application doesn't change the

SUMMARY OF NEW API FEATURES IN LASERWRITER 8

These are new selectors to PrGeneral. Full details and sample code are located on the CD. Remember, don't even *think* about requiring these calls in your program.

- PSPrimaryPPDOp: Returns an FSSpec record locating the PPD file chosen for this printer, along with a Boolean value indicating whether the file is the built-in "generic" PPD.
- getPSInfoOp: Informs you if the target printer supports PostScript Level 2 and binary communications. Also tells you what kind of PostScript file the user chose to create, if any: EPS with no preview, EPS with standard preview, EPS with enhanced preview, or PostScript job file format.
- PSIntentionsOp: Allows your code to tell the driver that you intend to use PostScript Level 2 or binary communications features, so that the driver can generate appropriate DSC (document structuring conventions) comments and return errors if the target printer doesn't support those features.
- PSpict2eps: Converts a QuickDraw picture into an EPS stream. You can specify most of the parameters in the style and job dialogs, and you provide a callback routine that receives the EPS stream as it's created.
- PSFontInfo: Provides a PostScript stream to make a given outline font available on a PostScript interpreter, including the TrueType scaler and TrueType conversions of fonts if necessary. You provide a callback routine to receive the font information from the driver. Bitmap fonts are not supported.

Document structuring conventions are discussed in Appendix G of the *PostScript Language Reference Manual,* second edition.•

clipping rectangle for the image it's printing and when you send no QuickDraw code, but only PostScript code. The driver watches all drawing you perform in the printing grafPort to calculate the bounding rectangle for the EPS file; it has no way of knowing what that rectangle would be for any PostScript code you send, so it relies on the clipping rectangle when your PostScript code is sent through the printing grafPort. Use ClipRect to set the clipping rectangle of the printing grafPort to match the height and width of your image. The driver will accumulate all your clipping rectangles and make the bounding rectangle of the EPS file the smallest rectangle that encloses all of them plus the bounding rectangles of all QuickDraw drawing you performed. If you send both QuickDraw and PostScript code as recommended, you won't have this problem.

OLD THINGS THAT ARE DIFFERENT NOW

Some things are bound to change when you rewrite code from the ground up — mostly implementation details that were never guaranteed. That doesn't stop enterprising programmers from finding such details and using them, though, and that's where a lot of compatibility problems occur. Here are some things that have changed in the new driver that shouldn't have affected your programs, but might have anyway.

Using private PostScript operators. Apple has advised programmers for a long time not to use "Laser Prep" or "md" dictionary operators — private PostScript operators used by the LaserWriter driver to get its work done. Those operators were never documented or guaranteed to work, and as the driver changed over the years so did many of the procedures, breaking applications that relied on them. It's no big surprise that almost every one of those operators is either gone or changed in the new driver.

There's been some confusion in the past about just what Apple was trying to warn against, so I'm pleased to take this opportunity to make it very clear: *If it's not in the PostScript Language Reference Manual, including appropriate supplements for your printer, don't use it.*

Don't use *any* PostScript procedures you didn't define unless they're part of the language. If you find a procedure defined in LaserWriter 8 that does something you want to do, don't call it! The PostScript system in QuickDraw GX is entirely different from the one in LaserWriter 8; if you just substitute one set of procedures that you shouldn't call for another, your application will break again in the near future.

I would even avoid using printer-specific features, because with EPS files you're never sure what the target printer will be. If you must use these features, wrap them in PostScript's "stop" mechanism so that the job will complete even if the feature creates an error.

Precision Bitmap Alignment. In version 8.0, the Page Setup option "Precision Bitmap Alignment" still says "(4% reduction)" at the end of the item in the Options dialog. That will change in versions after 8.0 because the code will change.

In 8.0 and earlier, the Precision Bitmap Alignment option simply reduces the coordinate system to 96% of the former value, turning 300-dpi printers into 288-dpi printers. Since 288 dpi is an even multiple of 72 dpi, the screen resolution, this prevents rounding errors in bitmaps where some bits would be slightly larger than others. Unfortunately, this only solves the problem on 300-dpi printers. On 400-dpi printers (such as are often found in Japan), it changes the resolution to 384 dpi, which doesn't help.

Sometime after 8.0, "Precision Bitmap Alignment" will start aligning to the nearest lower multiple of 72 dpi available on the target printer, calculated by the PostScript code when printing. On a 400-dpi printer, that's 360 dpi for a 10% reduction. On a 600-dpi printer, it's 576 dpi and is again a 4% reduction. The point is to get precisely aligned bitmaps, not to reduce by exactly 4%, and past 8.0 that's how it will work.

Font substitution. In the past, turning on fractional font widths with SetFractEnable(TRUE) disabled font substitution. With LaserWriter 8, you always get font substitution when you ask for it.

In LaserWriter 7.1.1 through LaserWriter 7.2, turning off font substitution would sometimes give you a TrueType version of a font if you had it — even if the printer had a Type 1 version available — depending on whether you'd turned off line layout or enabled fractional font widths, and on what day of the week it was. This inconsistent behavior is removed from LaserWriter 8: once again, Type 1 fonts are always picked over TrueType fonts if both are available to the printing system. This is largely for compatibility with pre-TrueType systems.

Older customization resources. The older drivers supported mechanisms for adding device-specific items to non-Apple printers, such as custom page sizes and code to enable sheet feeders. These resources, if present, are ignored by LaserWriter 8 — printer manufacturers can control feeders, custom page sizes, and other device-specific features through PPD files. We later discovered that some enterprising developers were using the 'feed' resource to control things other than sheet feeders, since the mechanism gave them a chance to execute code on the Macintosh during the job dialog. That wasn't what it was designed for, so no one made any effort to make sure that would still work. That mechanism is now gone.

Color QuickDraw pixel patterns. Before LaserWriter 8, attempting to print with pixel patterns (as opposed to older black-and-white patterns) generally produced stunning black blobs on your page. With LaserWriter 8, printing with pixel patterns works just as you'd hope it would if the user chooses either "Color/Grayscale" or "Calibrated Color/Grayscale" (making the printing grafPort a cGrafPort) and if the printer supports PostScript Level 2 so the driver can use the Level 2 PostScript pattern mechanism. On Level 1 devices, pixel patterns are implemented using screens (color screens if available, halftone screens otherwise) and the patterns are clipped to their upper left eight-by-eight grid, matching the dimensions of a regular QuickDraw pattern. Even though the implementation isn't perfect on Level 1 devices, something much closer to your desired pattern than a black blob now appears.

Preferences file. The new driver keeps a preferences file in the Preferences folder (or in the System Folder under System 6). It's in this file that the driver stores the parsed PPD data, plus the recorded choices of which printers match which PPDs and some other driver-specific data that needs to stick around. If you ever update drivers manually, delete the old preferences file. Its contents *will* change from version to version. If you share one driver file among systems trying to isolate or reproduce a problem and you want to have the best chance of seeing the problem again, be sure to keep the preferences file with the driver as you bop between systems.

THE REST IS WAITING FOR YOU

If you have comments or bugs to report about the LaserWriter 8 driver, you can send them on AppleLink to the read-only address LWDRIVER.BUG. You won't get a response or an acknowledgment, but your feedback will get to the people responsible for making changes. Don't forget to look at the information on the CD for the complete story!

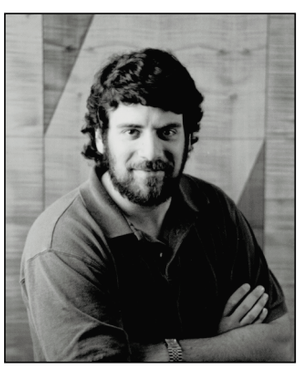
RECOMMENDED READING

- *Inside Macintosh* Volume II (Addison-Wesley, 1985), Chapter 5, and Volume V (Addison-Wesley, 1988), Chapter 22.
- *Inside Macintosh: Imaging With QuickDraw* (Addison-Wesley, forthcoming).
- "Meet PrGeneral, the Trap That Makes the Most of the Printing Manager" by Pete ("Luke") Alexander, *develop* Issue 3.
- "Print Hints: Top 10 Printing Crimes" by Pete ("Luke") Alexander, *develop* Issue 10.
- "Print Hints: Top 10 Printing Misdemeanors" by Pete ("Luke") Alexander, *develop* Issue 12.
- Macintosh Technical Note "Picture Comments — the Real Deal" (QuickDraw 10).

Thanks to Waymen Askey, Richard Blanchard, and Steve Winters for reviewing this column.•

COLOR MATCHING MADE EASY WITH QUICKDRAW GX

Accurate color matching used to be out of reach for most programmers to add to their applications. Then along came ColorSync, a system extension that provided a platform in QuickDraw for maintaining consistent color from device to device. Now with the advent of QuickDraw GX, which integrates ColorSync, color matching has been made even easier. Read on to find out how color matching works in QuickDraw GX and how to take full advantage of it, whether you're developing an application or a printer driver.



DANIEL LIPTON

Remember how impossible it used to be to get color images to display faithfully on any monitor or to print on any printer in colors that matched what the user saw on the screen? The introduction of ColorSync finally transformed color matching in QuickDraw from a complicated guessing game into a more-or-less predictable process. ColorSync provided a standard API that could produce WYSIWYG color output if used by both the application and the printer driver.

QuickDraw GX goes a step farther, fully integrating ColorSync for color management. When you create a QuickDraw GX application or printer driver, you don't need to worry about making ColorSync calls. You simply use the QuickDraw GX API and let it call ColorSync as appropriate. ColorSync does the work of converting a QuickDraw GX color specification into terms understandable to the output device. The first part of this article, which assumes you're somewhat familiar with QuickDraw GX, describes color specification and outlines the conversion process. What's left for your application or printer driver to do is the topic of the second part of this article.

COLOR SPECIFICATION IN QUICKDRAW GX

The starting point for color matching in QuickDraw GX is the exact specification of color that's included in every object to be drawn. Recall that the basic building block of QuickDraw GX graphics is the shape, an object that in turn points to other objects

DANIEL LIPTON works on QuickDraw GX and writes for *develop* to subsidize his songwriting career. He was recently overheard singing the following parody of the Steely Dan song "FM":

Bury your cubics, mama, quadratic's fine.
Kick off your PostScript printers, it's GX time.
The drivers don't seem to care what's where,
as long as the profile's there.

Nothing but greens and blues
and somebody else's favorite hues.
Give us some pumped-up colors, we'll sync them nice.
Feed us some hungry halftones, we'll print them thrice.
The printers don't seem to care what's where,
as long as GX is there.

No hassle at all.
GX. No hassle at all.•

that tell how it should be rendered. In particular, the ink object contains detailed information about the color to be used.

The color information for a geometric or typographic shape is contained in the gxColor data structure, which in abbreviated form looks like this:

```
typedef struct {
   gxColorSpace      space;           // the color space
   gxColorProfile    profile;         // the color profile
   union {
      . . .
      gxColorValue   component[4];    // the color value (one, three,
                                      // or four color components)
   } element;
} gxColor;
```

The color information for a bitmap shape is handled in a slightly different way, discussed at the end of this section. For any QuickDraw GX object, the color space, color profile, and color value together constitute a device-independent description of the color in which the object is to be rendered.

THE COLOR SPACE AND COLOR VALUE

A color space is a system for specifying colors. The color space determines how many different components are required to specify a color and what those components are. For example, the RGB color space uses three components to specify a color (red, green, and blue); the CMYK color space uses four (cyan, magenta, yellow, and black). The color value is the set of components that together specify a color. In QuickDraw GX, a color value consists of one, three, or four 16-bit integers, which are interpreted based on the color space. For instance, for a color specified in the RGB color space, the color value might be red = 32,768, green = 16,384, blue = 8,192.

In QuickDraw, all color is defined in the RGB color space. Unfortunately, since RGB is a device-dependent color space (more on this later), the same RGB color can look different on different devices. ColorSync provides a mechanism to match colors accurately, but QuickDraw applications are still restricted to RGB. High-end desktop publishing and photo-editing applications allow their users to work in other color spaces, but such applications have to include large chunks of code to work around QuickDraw in order to do this.

QuickDraw GX, on the other hand, provides a wide choice of color spaces. These color spaces can be grouped into three families: RGB, CMYK, and CIE. Within a family, one color space can be converted to another relatively simply. Color spaces in the RGB and CMYK families are device dependent because they're related to how a particular device represents color. Color spaces in the CIE family, on the other hand, are device independent because they're related to human visual perception.

For background information about QuickDraw GX, see these articles in *develop* Issue 15: "Getting Started With QuickDraw GX" by Pete ("Luke") Alexander; "Developing QuickDraw GX Printing Extensions" by Sam Weiss; and "QuickDraw GX for PostScript Programmers" by yours truly.•

How ColorSync works with QuickDraw is explained in John Wang's column "Print Hints: Syncing Up With ColorSync" in *develop* Issue 14.•

The RGB family of color spaces. Color spaces in the RGB family are based on controlling the intensities of red, green, and blue light, the three primary colors used in displays. Most desktop scanners and monitors as well as some printers work in some form of this color space. The RGB family consists of gxRGBSpace (red, green, blue), gxHLSSpace (hue, lightness, saturation), gxHSVSpace (hue, saturation, value), and gxGraySpace (a one-component gray scale).

The CMYK color space. The CMYK color space (the only member of the CMYK family) is based on controlling the concentrations of cyan, magenta, yellow, and black inks, the four process colors used in printing. While colors in the RGB color space are formed by adding light sources, colors in the CMYK color space are formed by subtracting light from an illuminating source. A component in a CMYK color value specifies the amount of light one of the inks absorbs. In theory, cyan absorbs red light, magenta absorbs green light, and yellow absorbs blue light.

Under ideal circumstances, mixing cyan, magenta, and yellow inks together on paper would produce a true black. However, due to ink impurities and a multitude of other problems, the result is usually a muddy dark brown. For this reason most ink-based devices have black ink as well.

The CIE family of color spaces. Color spaces in the CIE family are based on a three-component system of color specification developed by the Commission Internationale de l'Eclairage (CIE) in 1931. These color spaces are device independent because the color components are based not on intensities of light in a display or concentrations of printer inks but on aspects of how the human eye responds to light at different wavelengths. The CIE family of color spaces consists of gxXYZSpace, gxCIESpace, gxLUVSpace, gxLABSpace, and gxYIQSpace.

- The gxXYZSpace (XYZ color specification) and gxCIESpace (xyY specification) correspond to the 1931 Commission Internationale de l'Eclairage color description.
- The gxLUVSpace and gxLABSpace are transformations of the XYZ components that provide a more perceptually uniform color space. That is, throughout the three-dimensional space defined by the three components, a given distance moved numerically yields a constant perceptual change in the color.
- The gxYIQSpace is based on the U.S. standard video broadcast format defined by the NTSC.

THE COLOR PROFILE

Because of the variations in color representation among individual devices, simply specifying a color space and a color value doesn't provide enough information for color matching. The 50% red produced by an Apple 13-inch monitor's cathode ray tube, for example, doesn't look the same as the 50% red produced by a PowerBook

For color theory arcana, see the indispensable *Fundamentals of Interactive Computer Graphics* by J. D. Foley and A. Van Dam.•

180c's active matrix color display. Therefore, we also need to provide information in absolute terms about how colors look on the device on which an object is drawn. The color profile provides this information.

A color profile is a characterization of a device associated with an object, usually the device on which the object was created. The exact contents of the color profile depends on the color matching method to be used, but certain header data required by Apple's default color matching method is always present. This header data includes the device type, manufacturer, model, and, most important, an absolute description of each of the primary and secondary colors the device can render.

This absolute description consists of a set of response curves and chromaticities. The response curves are used to convert the color component values into linear values. The chromaticities are XYZ triplets describing the device's red, green, blue, cyan, magenta, yellow, black, and white. Recall that the XYZ color space is device independent; thus, an XYZ triplet describes a color in absolute terms.

SPECIFYING COLOR FOR BITMAP OBJECTS

While it would be possible to have a full color specification for every pixel in a bit image, this is neither practical nor necessary. Colors in bitmap objects (whether the object is a bitmap shape or part of a view device object) are handled slightly differently from colors in geometric and typographic shapes. The bitmap object contains a single color space and profile. Each pixel in the bit image contains a packed form of the color component values.

- For a 16-bit RGB bitmap, the high bit is ignored, followed by five bits of red, five bits of green, and five bits of blue.
- For a 32-bit RGB bitmap, the high eight bits are ignored, followed by eight bits of red, eight bits of green, and eight bits of blue.
- For a 32-bit CMYK bitmap, eight bits of cyan are followed by eight bits of magenta, eight bits of yellow, and eight bits of black.
- For all of the other 32-bit bitmaps (HLS, HSV, XYZ, CIE, LAB, LUV, YIQ), the high two bits are ignored, followed by ten bits per component.

Whether a component is expressed with four, five, eight, or ten bits, the bits in the data are the most significant bits of the standard 16-bits-per-component QuickDraw GX colors.

WHERE COLORSYNC COMES IN

As stated earlier, QuickDraw GX uses ColorSync for color management. Basically, QuickDraw GX calls ColorSync to do the necessary conversion from a source color to a matching destination color based on the color specification for a QuickDraw GX

object (which includes the color profile for the source device) and the color profile for the output device. Because QuickDraw GX calls ColorSync, your application doesn't need to.

ColorSync uses two basic elements to perform color matching: a color profile and a color matching method (CMM). The color profile, as you know, contains the device characterization, while the CMM is a component that contains code to perform the matching. Some CMMs are better than others, and some are more appropriate for certain kinds of devices (for example, ink jet printers versus dye sublimation printers).

A system will have at least one color profile for each device to be drawn on and at least one CMM to perform the matching. ColorSync comes with one Apple CMM (the default) and with color profiles for all Apple monitors currently being manufactured. A device can have more than one color profile, but only one is selected for use at any given time. The color profile specifies the CMM to be used. ColorSync will try to use this CMM, but if it's not available, will use the default Apple CMM. ColorSync's open architecture allows third-party developers to create their own profiles and CMMs if they want to perform matching beyond the capabilities of the Apple CMM.

In highly simplified terms, here's how the conversion process works, assuming the Apple CMM is used:

1. An application makes a QuickDraw GX call to draw an object.
2. QuickDraw GX calls ColorSync.
3. Using the object's color specification and the Apple CMM, ColorSync converts the color to the device-independent XYZ color space.
4. Using the Apple CMM and the color profile provided by the driver for the output device, ColorSync converts the color from the XYZ color space to the output device's color space.
5. QuickDraw GX draws the object on the output device.

If a CMM other than the Apple CMM is used, steps 3 and 4 may be different, since the CMM determines exactly how the conversion is done.

Now that you have a basic grasp of the mechanism for color matching, we'll turn to a consideration of what applications and printer drivers need to do to take full advantage of this mechanism.

WHAT AN APPLICATION NEEDS TO DO

If you're developing a QuickDraw GX application, there's really not much to worry about with respect to color matching. If you create your objects with appropriate

ColorSync version 1.0.3, along with documentation and samples, can be found on this issue's CD. The documentation describes how to create a CMM.•

color profiles, the colors will be rendered correctly to the extent that the output device is capable of rendering the specified colors. You can make life easier for your users by warning them when a color they choose can't be rendered on a designated printer and by enabling them to preview what a color would look like on a designated printer. And you can turn color matching off to improve performance in certain situations. We'll look at these techniques one at a time.

CREATE AND MANIPULATE A COLOR PROFILE

As we've seen, the key to color matching is providing an appropriate color profile for every QuickDraw GX shape object your application creates. If the objects created by your application are associated with a particular device, that device's color profile is the one to reference in the gxColor data structure. So if you're writing a scanning application, the bitmap objects you create should reference the color profile of the scanner used. If you're writing a painting or drawing application, your objects should reference the color profile of the monitor on which the objects were created.

It's appropriate to set the color profile to nil when dynamically creating objects that aren't associated with any particular device. When the color profile is set to nil, QuickDraw GX assumes the profile to be the default color profile. The default color profile is a color profile for some particular, perhaps imaginary, device. What the device is doesn't really matter because when any object is drawn, its color is automatically converted into the color space of the destination device using the destination device's color profile.

Applications can easily find out what a device's color space and color profile are. Every QuickDraw GX view device object contains a bitmap structure that contains color space and color profile fields. Files containing color profiles for particular monitors or scanners can often be found in the ColorSync Profiles folder, which is in the Preferences folder in your System Folder. Your code can get the ColorSync Profiles folder by calling GetColorSyncFolderSpec. The application almost never needs to know what the color space and profile of a printer are, because objects are seldom created from a printer, but there *is* a way to obtain this information, as described in the next section.

Given a set of ColorSync profile data, your application can create and manipulate a QuickDraw GX color profile object with the following functions. In all cases, the data is treated as a ColorSync profile and the ColorSync structures for profiles can be used.

```
gxColorProfile GXNewColorProfile(long size, void *data);
```

Creates a new color profile object with the data passed in. The size parameter is the size of the data, and the data parameter is a pointer to the data. The function result is the color profile object. If the size is 0, color matching will be disabled for those objects associated with this color profile object.

```
gxColorProfile GXSetColorProfile(gxColorProfile theProfile, long size,
   void *data);
```

Changes the data stored in the color profile object passed in the first argument. The size parameter is the size of the data, and the data parameter is a pointer to the data. The function result is the changed color profile object.

```
long GXGetColorProfile(gxColorProfile theProfile, void *data);
```

Retrieves the color profile data out of a color profile object. The function result is the size of the color profile data.

Here's how to get the ColorSync profile data from a color profile object:

```
size = GXGetColorProfile(myProfile, nil);
if (size > 0) {
   myPtr = NewPtr(size);
   GXGetColorProfile(myProfile, myPtr);
} else {
   /* Size = 0, indicating color matching should be suppressed. */
}
```

This function disposes of a color profile object:

```
void GXDisposeColorProfile(gxColorProfile theProfile);
```

CHECK TO SEE IF A COLOR IS IN GAMUT

Not all colors can be rendered on all devices. Each device has a set of colors that it's capable of reproducing, called a *color gamut*. When a color can't be duplicated on a device, the color is said to be "out of gamut" for that device (see Figure 1). How out-of-gamut colors are treated depends on the CMM being used: some CMMs may try to preserve the luminance of the color while others may try to preserve the hue or the saturation or some other aspect.

Given this state of affairs, you may want your application to warn users when a color they choose is out of gamut for the printer their document is currently formatted for. QuickDraw GX provides the following call to check a color against a device's color profile to see if it's in or out of gamut for the device:

```
Boolean GXCheckColor(gxColor theColor, gxColorSpace theColorSpace,
   gxColorSet theColorSet, gxColorProfile theProfile);
```

This call takes the gxColor (which contains a color profile and a color space) for the object, and the color space, color set (similar to a QuickDraw CLUT), and color profile for the device. The function returns true if the specified color can be rendered on the device and false if it can't.

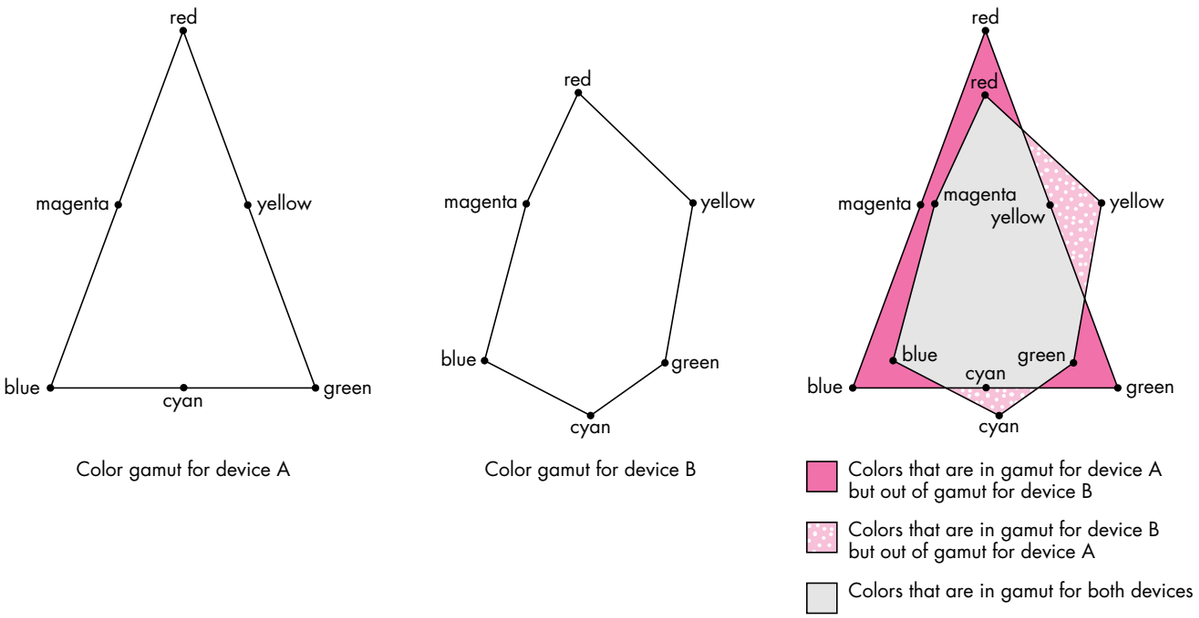


Figure 1
How the Color Gamuts of Two Different Devices Compare

It's a simple matter to obtain the color profile and color space for the printer a document is formatted for. Recall that for each printed document, there's a corresponding QuickDraw GX job object. The job object contains global document properties, such as the device information and the number of pages or copies. The device information is what we're after. Here's the code that gets it for us:

```
gxColorProfile GetFormattingPrinterProfile(gxJob myDocumentJob,
                                           gxColorSpace *theSpace)
   {
      gxPrinter      frmtPrinter;         // the formatting printer object
      gxViewDevice   printerDevice;       // the printer's view device
      gxShape        devBitmap;           // the device bitmap shape
      gxBitmap       devBits;             // the bitmap structure

      /* Get the bitmap shape for the printer's device. */
      frmtPrinter = GXGetJobFormattingPrinter(myDocumentJob);
      /* Pass in 0 as the index to obtain the currently selected view
         device from the driver's list of possible view devices. */
      printerDevice = GXGetPrinterViewDevice(frmtPrinter, 0);
      devBitmap = GXGetViewDeviceBitmap(printerDevice);
```

```
      /* Get the bitmap struct, dispose of the shape, return the
         profile. */
      GXGetBitmap(devBitmap, &devBits);
      GXDisposeShape(devBitmap);
      *theSpace = devBits.space;
      if (*theSpace == gxIndexedSpace)
         GXGetColorSet(devBits.set, theSpace, nil);
      return (devBits.profile);
   }
```

To obtain just the color profile (instead of the color profile *and* the color space, fetched by the preceding code), you can call GXFindPrinterProfile. The prototype is as follows:

```
long GXFindPrinterProfile(gxPrinter, void *searchData, long index,
   gxColorProfile *returnProfile)
```

PREVIEW AN OUT-OF-GAMUT COLOR

Using the printer's color profile, your application can also enable users to preview what an out-of-gamut color (or whole picture) would look like on that printer:

```
Boolean MakePrinterColor(gxJob theJob, gxColor *sourceColor,
                         gxColor *printedColor)
   {
      gxColorProfile    printerProfile;
      gxColorSpace      printerSpace;
      Boolean           inGamut;

      /* Get the printer's profile. */
      printerProfile = GetFormattingPrinterProfile(theJob, &printerSpace);
      /* Copy the source color. */
      *printedColor = *sourceColor;
      /* Check it and convert it into the device's color space. */
      inGamut = GXCheckColor(printedColor, printerSpace, nil,
                             printerProfile);
      GXConvertColor(printedColor, printerSpace, nil, printerProfile);
      return (inGamut);
   }
```

The color passed into this routine is converted into the printer's color space and profile. The most closely matching color from the printer's gamut is converted back to the screen's color space and profile when the color is associated with a shape and drawn on the screen. Thus, a simulation of what the printer's output would look like is achieved. As a function result, the code returns whether the color is in or out of gamut for the printer.

To preview an entire picture, set up an off-screen bitmap in the printer's color space and color profile, set the gxEnableMatchPort attribute (explained in the following section) of the view port you're using with the off-screen bitmap, draw the picture into that off-screen bitmap, and then draw that bitmap on the screen. Make sure also that the view port you're using to draw on the screen has gxEnableMatchPort set.

TURN COLOR MATCHING OFF AND ON

Color matching is a computationally intensive process, so it slows down performance. In some situations, such as during scrolling or updating, you may be willing to sacrifice accurate color in exchange for faster drawing. QuickDraw GX enables you to turn off color matching in these situations, either for all objects drawn into a view port or on an object-by-object basis.

You can control color matching for all objects drawn into a view port with an attribute of the view port object called gxEnableMatchPort. When this bit in the view port's attributes is set (using GXSetPortAttributes), color matching is performed for all shape objects drawn into that view port. When this bit is cleared, the color matching process is bypassed. The result is less-than-WYSIWYG output, but the drawing is faster. Note that the view port's default is to bypass color matching; your application has to set the bit to turn color matching on.

You can control color matching on an object-by-object basis by creating a color profile object of length 0 and associating it with those objects you want to disable matching for. If an object has the zero-length profile, it isn't matched, even if the view port's gxEnableMatchPort attribute is set.

WHAT A PRINTER DRIVER NEEDS TO DO

If you're developing or thinking of developing a QuickDraw GX printer driver, you know that it's radically easier than developing a QuickDraw driver. Color matching is easier with a QuickDraw GX driver as well. In the old world of QuickDraw, your driver had to have special code to call out to ColorSync if it wanted to do color matching. In a QuickDraw GX driver, you don't have to call anything to get color matching for your printer. All you need to do is specify at least one color profile.

If you're developing a printer driver for a PostScript device, there are some things you should know to obtain the highest quality color output from your printer, whether it's a Level 1 black-and-white or color printer or a Level 2 color printer. Specifically, with fields in the data structure that gxPostScriptImageDataHdl points to, you can choose a color space, offload color matching to a Level 2 device, or generate PostScript code that's Level 2 savvy but can also run on a Level 1 printer (color or black and white) while retaining all the color information in the source data.

Incidentally, most things discussed here can be implemented in a printing extension as well.

The levels of PostScript and all other details you could possibly want to know about PostScript are described in the *PostScript Language Reference Manual,* second edition.•

PROVIDE AT LEAST ONE COLOR PROFILE

Although only one color profile is used at a time in the color matching process, a printer can have more than one color profile. Each one can be associated with a particular format (recall that in QuickDraw GX, a format is an object containing the properties associated with a particular page, including the paper type, which is an object describing printing media). For instance, the Apple Color Printer has default color profiles for coated paper, transparency film, and plain paper; a different color profile is needed for each because paper type can affect the appearance of color.

If you have only one color profile for your printer, a simple and common case, you can store the profile data in a 'prfl' resource in your printer driver. QuickDraw GX will read in the data from the resource (using the default implementation of the GXFetchTaggedData message), make a color profile object out of it, and automatically associate it with your printer. If you want to create the color profile dynamically rather than store it in a resource, just override the GXFetchTaggedData message, looking for the tag 'prfl' and creating the handle on the fly.

Applications can query your printer driver with a GXFindFormatProfile call to find out which color profile will be used for a particular page of output. (Within the same document, different pages can be printed on different paper types. For example, a business letter document might contain an address that prints on an envelope, a letter that prints on white paper, and a resumé that prints on blue paper.) To support this application query, your driver must override two messages:

- GXFindFormatProfile, which is normally sent in response to the application's call. Override this message and return to the application the profile that would be used with the specified format object.
- GXImagePage, which is normally sent before imaging each page of the document. Override this message to set the color profile on a page-by-page basis. Your override will be passed a format object, which will contain a paper type object from which you can determine (and create, if necessary) the appropriate color profile to use. Your override will also be passed an image data handle. One of the fields in the structure that this handle points to is a color profile. Simply set this to be the profile you want and forward the message, and QuickDraw GX does the rest.

CHOOSE A COLOR SPACE

The preceding discussion of color profiles holds true for all three classes of QuickDraw GX printer drivers: raster, vector, and PostScript. The remaining discussion applies only to printer drivers for PostScript devices.

PostScript code can describe colors that the output device is to produce in any of three different device color spaces. In each case, different operators are used. When

The messaging scheme for drivers and extensions is described in Sam Weiss's article "Developing QuickDraw GX Printing Extensions" in *develop* Issue 15, and in *Inside Macintosh: Printing Extensions and Drivers*.•

you set the color space, you tell QuickDraw GX what kind of PostScript operators to use when specifying color for your printer, based on its color capabilities.

You set the color space in the field devCSpace (of type gxColorSpace) in the PostScript image data structure. Only three values are allowed:

- gxRGBSpace, which tells QuickDraw GX to use the **setrgbcolor** and **colorimage** operators in the PostScript language
- gxCMYKSpace, which tells it to use the **setcmykcolor** and **colorimage** operators
- gxGraySpace, which tells it to use the **setgray** and **image** operators

QuickDraw GX calls ColorSync to convert all the colors to be printed into the specified color space using the color profile your driver provides. It's up to you to specify values that make sense for your printer, as QuickDraw GX does no sanity checking. For example, if you specify gxCMYKSpace as your color space but connect to a printer on which the **setcmykcolor** operator isn't available, you'll get PostScript errors. The only color space guaranteed to work on all PostScript printers is gxGraySpace. You can get around this problem by generating portable PostScript code, discussed later.

OFFLOAD COLOR MATCHING

The PostScript Level 2 interpreter has color matching support built in. This means that you can offload the expensive work of color matching from the Macintosh to the PostScript device if it has a Level 2 interpreter. To take advantage of this, set fields in the PostScript image data structure as follows:

- Set the languagelevel field to 2.
- Set the gxUseLevel2ColorOption bit in the renderoptions field.

With these settings in effect, QuickDraw GX generates PostScript code that's optimized for PostScript Level 2 and uses the color management provided by the Level 2 interpreter instead of calling ColorSync. The color space and color profile of the objects to be printed are translated into a Level 2 color space dictionary, using the **setcolorspace** operator. The colors for objects are then set in the graphics state using the **setcolor** operator, and bitmaps are drawn using the dictionary form of the **image** operator. The **image** operator is used at eight bits per component when the source bitmap's color space is a 5- or 8-bits-per-component space, and 12 bits per component when the source bitmap's color space is a 10-bits-per-component space.

If the gxUseLevel2ColorOption bit isn't set but the language level is 2, QuickDraw GX will generate code optimized for Level 2 but will work with color based on the devCSpace as explained earlier.

Not all QuickDraw GX color spaces can be translated to Level 2. For a color space that can't, QuickDraw GX performs a conversion into one that can. For example, gxCIESpace (the CIE xyY space) can't be emulated with the **setcolorspace** operator. All colors in gxCIESpace are converted into gxXYZSpace — a color space that can be emulated with the **setcolorspace** operator.

GENERATE PORTABLE POSTSCRIPT CODE

Sometimes you don't know what kind of PostScript device your code is going to end up on. Because all PostScript printers answer to "LaserWriter," a user can connect to just about any kind of PostScript printer with your QuickDraw GX printer driver. But, as mentioned earlier, if your driver specifies a color space that's not available on the printer the user connects with, this will generate PostScript errors. To avoid this situation, QuickDraw GX is capable of generating "portable" PostScript code — code that can be executed on any printer and will produce the best results that printer is capable of, although it's not necessarily optimized for any one printer.

As stated earlier, the only color space guaranteed to work on all PostScript printers is gxGraySpace. However, using this color space causes output to be grayscale even if the PostScript code is sent to a color printer. To get QuickDraw GX to produce PostScript data that contains all color information but will also render on a black-and-white PostScript device in grayscale, set the gxPortablePostScriptOption bit in the renderoptions field and set the devCSpace field to gxRGBSpace.

When you do this, QuickDraw GX defines PostScript procedures to emulate the color operators when they're not present on the printer that the PostScript file lands on. Additionally, QuickDraw GX generates PostScript code to set up a Level 2 color space based on RGB and the color profile specified by the driver. When the PostScript file lands on a Level 2 color printer, you get color-matched output. The source colors are converted by ColorSync to the RGB color space using the driver's color profile. This color profile is translated into a **setcolorspace** operator so that those RGB colors have meaning. The translated color profile is ignored on Level 1 printers and the normal **setrgbcolor** and **colorimage** operators are used.

AN ILLUSTRATION: THE LASERWRITER GX DRIVER

Let's consider how the LaserWriter GX driver sets up the PostScript image data structure. When the driver is used to print to any of the current line of Apple PostScript printers, the data structure is set up as follows:

- The devCSpace field is set to gxGraySpace (because all Apple PostScript printers are black and white).
- The languagelevel field is set to 1 or 2 depending on the printer.
- The devCProfile field is set to nil.

When the LaserWriter GX driver doesn't recognize the printer it's talking to, the data structure is set up like this:

- The devCSpace field is set to gxRGBSpace.
- The gxPortablePostScriptOption bit is set in the renderoptions field.

This yields portable PostScript code, which is Level 2 savvy but can also run on a Level 1 color or black-and-white printer while retaining all the color information in the source data. Thus, using the LaserWriter GX driver (from either QuickDraw GX applications or QuickDraw applications) gives better and faster output for color images on Apple black-and-white printers, color printing on non-Apple Level 1 color printers, and color-matched printing on Level 2 color printers.

COLOR YOUR WORLD

Whether you're developing an application or a printer driver, color matching has never been easier than with QuickDraw GX. You can work in whichever color space you want and move data from device to device without worrying about losing information or writing special code to handle the conversions. Getting basic color matching is free (no code is necessary) and getting high-end tuned results is easy (only small amounts of code are required).

The QuickDraw GX color publishing platform seamlessly integrates high-end text and graphics with the capabilities offered by ColorSync and the PostScript Level 2 interpreter. And if you have a great color matching algorithm, you can easily integrate your method with all QuickDraw GX applications and printer drivers simply by writing a standard ColorSync color matching method and providing color profiles. Color no longer needs to be a complicated guessing game for the user.

REFERENCES

- "Getting Started With QuickDraw GX" by Pete ("Luke") Alexander, "Developing QuickDraw GX Printing Extensions" by Sam Weiss, and "QuickDraw GX for PostScript Programmers" by Daniel Lipton, *develop* Issue 15.
- "Print Hints: Syncing Up With ColorSync" by John Wang, *develop* Issue 14.
- *Fundamentals of Interactive Computer Graphics* by J. D. Foley and A. Van Dam (Addison-Wesley, 1982).
- *Inside Macintosh: Printing Extensions and Drivers* (Addison-Wesley, 1993).
- *PostScript Language Reference Manual*, 2nd ed., by Adobe Systems Incorporated (Addison-Wesley, 1990).

THANKS TO OUR TECHNICAL REVIEWERS Pete ("Luke") Alexander, Tom Dowdy, Dennis Farnden, Josh Horwich•

Special thanks to duaño, Sean Allen, Chris Yerga, and Dean Yu.•

GRAPHICAL TRUFFLES

REMEDIES FOR COMMON QUICKDRAW PROBLEMS

JOHN WANG

During the past two years alone, the Developer Support Center has answered more than 1000 QuickDraw-related questions. The answers to most of these questions are now available in Apple's plentiful "one-to-many" sources of support for developers, including *Inside Macintosh*, Technical Notes, Q&As, sample code, and of course *develop*. But you might find it helpful to look here first if you're having a QuickDraw problem. The symptoms of some common problems are listed in this column along with Dr. John's suggested remedies.

These are the symptoms that we'll suggest remedies for:

- CopyBits is too slow.
- You have a palette created with a tolerance of 0, but the colors in your graphics port don't match the palette.
- You're using palettes stored in 'pltt' resources and they don't seem to have any effect.
- Strange colors get drawn when you do a CopyBits between different graphics ports.
- The pen pattern isn't being used when PaintRect is called with hilite penmode.
- NewGWorld doesn't return an error and the GWorldPtr it returns is unchanged.
- QuickDraw routines aren't working on your GWorld.
- Your complement procedure isn't being called for InvertRect.

THE REMEDIES

CopyBits is too slow.

Remedy: There's little doubt that CopyBits is a complex piece of code, so there are *many* factors that can affect its execution speed. In fact, there's a long Technical Note dedicated to this topic, called "Of Time and Space and _CopyBits" (QuickDraw 21). But here are a couple of quick hints for possible ways to speed up CopyBits:

- If your source and destination graphics ports have matching color tables, set the ctSeed field in the color tables of the source and destination to be the same. This removes the overhead of comparing the entries in the color tables to determine whether color mapping is necessary.
- Use GWorlds when copying the entire off-screen buffer to the screen. GWorlds will properly align your pixel data so that CopyBits calls don't require byte and bit shifting. (You must create the GWorld with a depth of 0 and pass the rectangle in global coordinates.)

You have a palette created with a tolerance of 0, but the colors in your graphics port don't match the palette.

Remedy: The colors in the graphics port may appear to be different, but remember that only the high eight bits of an RGB color component are important. The lower eight bits aren't significant because they're ignored by the hardware. Consequently, for optimization and implementation reasons, NewPalette creates a palette of colors by copying the high byte of each color in the input color table to both the high byte and the low byte of the palette. For example, if the color table has the color ($ff00, $75fe, $0080), the equivalent palette entry would be ($ffff, $7575, $0000).

JOHN WANG (AppleLink WANG.JY) of Apple's Developer Support Center has a new addition to his family, named Pepper. The four-month-old baby girl weighs in at a healthy nine pounds. She has big brown eyes, baby teeth, and a generous amount of hair. She sleeps all day and makes almost no noise at night. And she's nearly toilet trained already! Pepper even gets along well with her older brother Skate; they like to nibble playfully on each other's ears. But when it comes to food, they're fearless; they fight, kick, and howl — all for just a bone.•

You're using palettes stored in 'pltt' resources and they don't seem to have any effect.

Remedy: You're having this problem because GetNewPalette doesn't work as documented in *Inside Macintosh* Volume VI. The description of this routine in Volume VI states that a palette will be loaded and attached to the current window, and if the palette requested isn't available, the default application palette is used instead. The actual implementation of GetNewPalette is much simpler: it only loads the specified 'pltt' resource with GetResource and detaches it with DetachResource to make it a handle; if the specified 'pltt' resource isn't found, GetNewPalette doesn't load the default application palette.

Strange colors get drawn when you do a CopyBits between different graphics ports.

Remedy: More than likely, your graphics port and device are incorrectly set. The current port and device must always be set to the destination port and device. So if you're copying from a window to an off-screen GWorld, you must call SetGWorld to set the GWorld as the current port and device. When you do a CopyBits from a GWorld to a window, you must set the port to the window and the graphics device to the MainGDevice. This rule actually applies to all QuickDraw drawing.

The pen pattern isn't being used when PaintRect is called with hilite penmode.

Remedy: A bug in QuickDraw causes the pen pattern to be ignored when used with hilite penmode in the following calls:

- FrameRect, PaintRect, and FillRect
- LineTo (vertical and horizontal lines only)
- FrameRgn, PaintRgn, and FillRgn (rectangular regions only)
- FramePoly, PaintPoly, and FillPoly (rectangular polygons only)

A simple workaround for PaintRect is to call PaintRoundRect instead; that is, call

```
PaintRoundRect(&myRect, 0, 0);
```

rather than

```
PaintRect(&myRect);
```

NewGWorld doesn't return an error and the GWorldPtr it returns is unchanged.

Remedy: The cause of this problem is typically memory movement. Many object-oriented languages, such as MacApp, store data in handles. If the GWorldPtr pointer variable you pass to NewGWorld is stored in a relocatable block of memory, and if that block moves during execution of NewGWorld, NewGWorld will store the GWorldPtr in the old dereferenced storage location. Instead, pass a local GWorldPtr to NewGWorld and copy the local GWorldPtr to the relocatable memory block afterward.

QuickDraw routines aren't working on your GWorld.

Remedy: Although some routines in QuickDraw will work when LockPixels isn't called, most routines will have unexpected results. Always call LockPixels if you want to access or draw into your GWorld. To prevent memory fragmentation, the GWorld image should be unlocked when you're not accessing the pixel data so that it can move in memory.

Your complement procedure isn't being called for InvertRect.

Remedy: Complement procedures are called from InvertColor only. All other Invert calls simply invert bits, as in QuickDraw's original design.

NEED A REAL DOCTOR?

The next time a QuickDraw problem is giving you a headache, look here first for help. Dr. John's list of remedies may provide just what you need to solve your problem. If you don't find a solution here, check as usual in *Inside Macintosh* and the many other available sources of help.

Thanks to Don Moccia and Brigham Stevens for reviewing this column.•

INTERNATIONAL NUMBER FORMATTING

Have you ever wondered how to get your program to display numbers in a way that satisfies Macintosh users all around the world? This article tells you what users expect and shows you how to use the Macintosh Toolbox to correctly format numbers, taking the needs of both your program and the user into account. It also shows how to interpret numbers entered by the user.

NORBERT LINDENBERG

When you develop an application, you usually have some opinion about the format in which numbers should be presented to the user. However, number formatting standards differ from country to country (and sometimes even within a country), and users also may have their own ideas on the subject. Macintosh system software provides support to format numbers in ways that accommodate both the needs of your application and local standards, and — starting with System 7.1 — also lets the user control some aspects of number formatting using the Numbers control panel.

This article shows two different ways to format numbers: using a default format for simple number display, and following the user's specification for more sophisticated number display. It also shows how to interpret numeric input correctly. This issue's CD contains an application called Numbers Test that lets you try out these two different methods of formatting numbers and enter numbers for interpretation. The CD also contains BuildNumbers, an MPW script that builds an MPW tool that's functionally equivalent to the application.

WHAT USERS EXPECT

Users expect to see numbers in a format that makes sense to them. This challenges the programmer to accommodate the variations on number formatting that occur around the world.

The most common system for writing numbers is the decimal system, where numbers are formed from ten different numerals, with the position of each digit within a number defining a multiplier for it: 123 = 1*100 + 2*10 + 3*1. However, there are

NORBERT LINDENBERG is an import from Germany who still wonders why there are offices in California that don't have windows and need artificial light while the sun is shining outside, and how Americans can survive on 12 days of vacation a year instead of the normal six weeks. While a student at the University of Karlsruhe, Norbert developed the Literate Programming Workshop, which he claims is the best environment for doing literate programming. (Literate programming is similar to a *develop* article in that you combine source code and documentation in one document, but different because you compile the source code right out of this integrated source document.) Now he actually gets paid for writing literate programs for Apple's International Software Support Group.•

many local variations on this scheme, and there are some writing systems that prefer a different style of writing numbers, in which case decimal numbers may or may not be an acceptable alternative. Systems besides the decimal system that users may require include Roman numerals (used in many languages to number topics or title pages) and hexadecimal numbers (familiar to everybody who's ever dropped into MacsBug), as well as the Japanese and Chinese systems. For details on how these number formatting systems differ from one another, see "Number Formatting Variations."

Computers may complicate matters even more by providing multiple character codes for the same digit. For example, the Macintosh Japanese character set provides both 1-byte and 2-byte encodings of the Latin characters (which are called "Romaji" in Japanese). They can easily be distinguished on the screen: the 1-byte versions are narrower than the 2-byte versions, which take up the same width as Kanji characters. For interpretation as numbers, however, these different encodings should be considered equivalent.

Another example is the Macintosh Arabic character set, which defines a set of Arabic digits with right-to-left orientation in addition to the ASCII digits, which have left-to-right orientation and are usually displayed with Arabic glyphs when an Arabic font has been chosen. The right-to-left digits are intended only for text that doesn't have a numeric meaning, such as software version strings and part numbers, and are needed to obtain proper line layout in these cases. However, users may not be aware of this intention and may try to enter numbers using these digits. Later we'll discuss how to deal with this.

WHAT YOUR APPLICATION NEEDS

Depending on how sophisticated your application is with regard to numbers, you'll need to support variations on number formatting in three different situations: simple number display, number display in a user-specified format, and numeric input.

For simple number display, your application needs to show a given number in a default format that makes sense to the user. This kind of formatting may suffice for many applications and is commonly used for dialogs.

For other number-display situations, your application might need to format numbers according to the user's specification. The user might specify which representation to use for the number (for example, decimal or traditional Chinese; Thai, Arabic, or Latin glyphs), the number of digits after the decimal separator, how to indicate negative numbers, whether to use thousands separators, which currency symbol to use, and where to place it. This kind of formatting is needed, for example, for spreadsheets, databases, and page layout applications.

Numeric input is needed in almost any application — for example, to specify the width of a page, the number of a page to jump to, or the size of a font. Ideally, your

For background information on number formatting, see Chapter 5, "Text Utilities," in *Inside Macintosh: Text.* For definitions of basic terms used in this article, see Chapter 1, "Introduction to Text on the Macintosh," in the same volume.•

NUMBER FORMATTING VARIATIONS

Local variations on the decimal system include variations on the shapes of the digits, representation of negative numbers, the decimal separator, and the thousands separator.

- The shapes of the digits: The glyphs used with the Latin writing system differ from those used with the Arabic writing system, and several other writing systems come with their own glyphs.

Latin	0	1	2	3	4	5	6	7	8	9
Arabic	٠	١	٢	٣	٤	٥	٦	٧	٨	٩
Thai	๐	๑	๒	๓	๔	๕	๖	๗	๘	๙

- Representation of negative numbers: The minus sign can be used before or after the number, or the number can be parenthesized.
- The decimal separator: Either a period or a comma can be used to mark off the integer part of the number from the fractional part.
- The thousands separator: A space, a comma, a period, or some other character can be used to mark off the thousands place from the hundreds place, the millions place from the hundred thousands place, and so on. Sometimes the thousands separator isn't used at all.

Many other variations exist, especially for noninteger numbers. Here's a sample of local variations on how one negative number is represented in the decimal system:

Arabic	١,٢٣٤,٥٦-
French	–1 234,56
German	–1.234,56
Greek	(1 234.56)
Japanese	(1,234.56)
Swiss French	–1'234.56
Thai	(๑,๒๓๔.๕๖)
U.S.	–1,234.56

In the Roman system, numbers are formed from letter digits representing the numbers shown below.

M	1000
D	500
C	100
L	50
X	10
V	5
I	1

Originally the digits of the number were simply added up to arrive at the value of the number, and digits were sorted in decreasing order within the number (so 9 = VIIII). Later a convention was added that positioning one of the digits C, X, or I before a higher-valued digit means that its value is to be subtracted instead of added (so 9 = IX).

The Japanese and Chinese systems represent numbers in various ways. In horizontal writing, the decimal system with Latin glyphs is commonly used. Ten thousands separators were once used instead of thousands separators and are still used in some very traditional quarters, but accountants in Japan now use thousands separators instead. In the traditional vertical writing preferred by native speakers, however, Chinese characters are used without separators. A mapping of decimal numbers to Chinese digits is acceptable; however, a direct representation of the numbers as they are spoken is preferred. The number 45000, for example, is represented in the decimal style on the left and in the traditional style on the right:

四	4	四	4
五	5	万	ten thousand
〇	0	五	5
〇	0	千	thousand
〇	0		

application should be able to interpret a numeric string in any format that might make sense to the user, independent of the display formats you use.

WHAT MACINTOSH SYSTEM SOFTWARE PROVIDES

Macintosh system software supports number formatting with international resources, the Numbers control panel (in System 7.1), and the Text Utilities routines. Unfortunately, the functionality provided doesn't cover all the needs just described — it's limited to decimal numbers and a maximum of two encodings per script. This means that, for instance, Chinese vertical numbers aren't supported; with the advent of QuickDraw GX, which supports vertical text, this problem is becoming more urgent. There are some interesting details you'll have to understand to make the best use of the functionality provided.

INTERNATIONAL RESOURCES

International resources of two types, 'itl0' and 'itl4', provide data that helps in formatting numbers.

- Resources of type 'itl0' contain separator symbols (decimal separator and thousands separator) and information about a simple default format. These resources allow for 1-byte characters only and don't support more sophisticated layout.
- Resources of type 'itl4' contain a number parts table used by the Text Utilities routines to interpret format specification strings entered or selected by the user. They also contain a table of alternate digits that can be used instead of the default ASCII digits and that may be 2-byte characters. If there are no alternate digits for the script, the ASCII digits are repeated in this table.

A system file can contain multiple resources of either type. Each regional version of system software comes with a default resource of each type, as well as the U.S. versions of the resources; more resources can be added.

If multiple scripts are installed on one machine, each script has at least one resource of each type and designates one resource of each type as the default for the script. The default resources for the system script (the script that supports the language your system is localized for) define the systemwide default. If GetIntlResource (IUGetIntl) is used to access a resource, the script whose resources are returned depends on the font in the current graphics port and the settings of the international resources selection flag. To avoid surprises, it's usually better to ask for resources of specific scripts; the InitializeDefaultNumberSeparators routine, discussed later, does this.

All Macintosh scripts support the use of the ASCII digits ($30–$39), and some scripts provide an additional set of digits in an alternate numeral table. The Japanese 'itl4' resource contains the 2-byte Romaji digits; the Arabic 'itl4' resource, the right-to-left

The number in international resource types isn't related to functionality. Resources of type 'itl1' contain long date strings; resources of type 'itl2' deal with text handling (sorting, uppercase and lowercase, word boundaries); and resources of type 'itl5' define rendering and character encoding. There are no resources of type 'itl3'.•

GetIntlResource or IUGetIntl? With the new edition of *Inside Macintosh*, many Toolbox routines have been renamed. However, interface files defining the new names are not yet available for all Macintosh programming environments. Therefore, I use the new names in the documentation, followed by the old names in parentheses, and I use the old names in the source code.•

digits; and the Thai 'itl4' resource, the Thai digits. Because only one alternate numeral table is allowed per 'itl4' resource, you won't find in the Japanese 'itl4' resource the Chinese numerals used in the Japanese script. Unfortunately, not all scripts that have multiple sets of digits define them in the 'itl4' resource; for instance, the Chinese versions of System 7.1 don't make use of the alternate numeral table but only support the ASCII digits.

THE NUMBERS CONTROL PANEL

The Numbers control panel (in System 7.1) lets users select the default number format and define customized decimal and thousands separators, as well as the currency symbol. In earlier systems, the International control panel (which was shipped only with certain localized versions of system software) allowed the user to select the default number format but didn't provide for customization. (See Figure 1.)

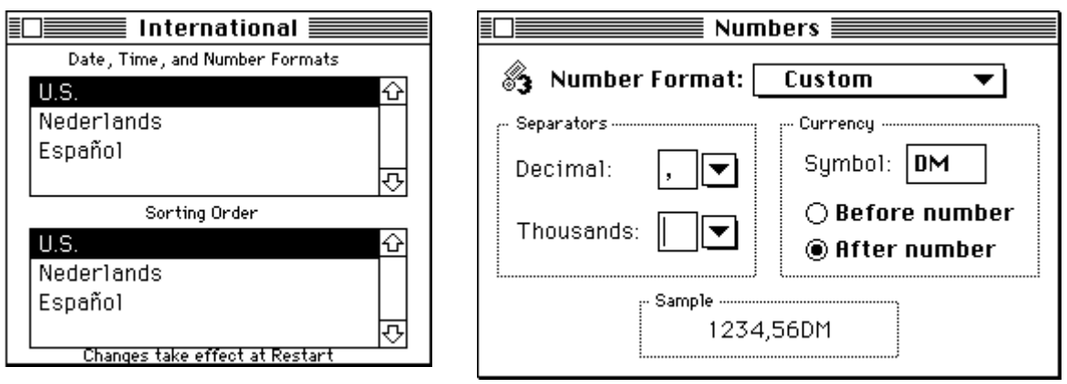


Figure 1
The International and Numbers Control Panels

To correctly access the international resources and interpret their contents, it helps to know how the control panels affect the resources. The behavior of the control panels has changed significantly from system software versions 7.0 and 7.0.1 to version 7.1. The International control panel in versions 7.0 and 7.0.1 lets the user select only the language whose number formatting rules apply; it does not allow modification of the rules. Selecting a language makes the corresponding region's 'itl0' resource the default resource for its script, so that all its features take effect. The 'itl4' resources are not affected.

The Numbers control panel in System 7.1 lets the user select a predefined regional version or define a custom version of the number format. The first time the control panel is opened after installing System 7.1, it creates a new 'itl0' resource in the System file based on the predefined default 'itl0' resource of this version of system

GetIntlResource (IUGetIntl) is described in *Inside Macintosh: Text,* pages 6-90 to 6-91. The international resources selection flag is described in *Inside Macintosh: Text,* pages 6-21 to 6-26.•

software and makes this new resource the default for the system script. From then on it keeps the user's format definition in this personalized 'itl0' resource, whether it's selected from predefined formats or defined as a custom format.

When the user selects a different regional version, all items of that region's 'itl0' resource that are represented in the control panel are copied into the personalized 'itl0' resource; other features defined in the 'itl0' resource are ignored. This means that the decision, for example, whether to show negative numbers with a minus sign or in parentheses is not affected by the selection. The default selection of the 'itl4' resource isn't changed; however, the default 'itl4' resource is modified to use the personalized 'itl0' resource's decimal and thousands separators in its number parts table.

There's one problem with the Numbers control panel that you have to be aware of: it doesn't impose any constraints on the selections for the decimal and thousands separators, other than not allowing the user to enter 2-byte characters. The user can, for example, select a digit, the minus sign, no character at all, or a character that conflicts with the inner workings of the Text Utilities routines for interpreting format specifications. To make sure that your application functions correctly, you have to check whether the separators make sense before using them.

The sample code discussed in this article assumes that you don't check for changes of the resources while your application is running, so it gets all necessary information at launch time and caches it. This way, changes made with the control panel will not be immediately reflected in your application, but you also avoid the problem of inconsistent updates. This problem can arise if you always use the most current information, and the user changes, say, the decimal separator while your application is displaying numbers in a window; in this case, it could happen that after redrawing a part of your window you display one decimal separator in the updated region and another one in the rest of the window.

THE TEXT UTILITIES ROUTINES

The Text Utilities routines format or interpret numbers according to format specifications that are given by format strings and that can be quite sophisticated so that context-dependent variations can be taken into account. Format strings are used in some spreadsheet and database applications and look like this:

```
'###.###,##;(###.###,##);0.##'
```

By default, the Text Utilities routines assume that numbers are encoded by ASCII digits ($30–$39) and displayed using Latin glyphs. However, we'll see that there's a way to have the routines support the set of digits defined in the alternate numeral table in the 'itl4' resource, in addition to ASCII digits.

The Text Utilities routines assume a localized format string. There's only limited support for automatically adjusting a generic format string to local customs or the user's preferences: the routines can replace characters by using a different number parts table but cannot convert to a different structure of the number format. For example, the indicator of negative numbers can't be switched automatically from the minus sign to parentheses. This is a problem if your application isn't localized for all the regions supported by Macintosh system software.

Now you know what users expect, what your application's needs are, and what support Macintosh system software offers when it comes to number formatting. You've seen that the Toolbox doesn't provide a solution for all your needs, so you'll have to extend it in some cases. We'll now look at how you can make the best use of what *is* provided to do simple number display using a default format, to display numbers in a user-specified format, and to interpret numeric input. The sample code presented here uses the international resources either directly or in combination with the Text Utilities routines to make up for at least some of the shortcomings of the Text Utilities routines.

SIMPLE NUMBER DISPLAY

First I'll show you how to use the 'itl0' resources in conjunction with the Text Utilities or Standard Apple Numerics Environment (SANE) routines to display numbers in the default number format. This method provides a simple solution for cases of simple number display.

The following code takes the localized or user-defined decimal and thousands separators into account. It assumes that numbers are written as integer or fixed-point decimal numbers in the ASCII character set and displayed in a font of the system script. It doesn't support Roman numerals, full-width Romaji, Chinese numbers, Thai digits, or the like. Negative numbers are written with a leading minus sign; parentheses aren't supported.

In some cases you won't want to use the default number formatting definition but instead will want to use the definition for a specific language. This case isn't taken into account in this version.

We start by defining the variables used to cache the default decimal and thousands separators. They must be initialized by calling InitializeDefaultNumberSeparators when the application is launched.

```
PROGRAM Numbers;
USES  Script, Resources, Memory, Errors, GestaltEqu, Packages, SANE,
      UFailure;
VAR   gDefaultDecimalSeparator:     Char;
      gDefaultThousandsSeparator:   Char;
```


SANE is the set of routines for floating-point calculations in Apple computers. It's documented in the *Apple Numerics Manual.*•

We call the procedure InitializeDefaultNumberSeparators in the application's initialization sequence to initialize both gDefaultDecimalSeparator and gDefaultThousandsSeparator from the default 'itl0' resource of the system script. If your application tracks changes in the Script Manager state, you can reinitialize the variables by calling InitializeDefaultNumberSeparators again. Because the Numbers control panel lets the user select any characters as the separators, we verify that the selection doesn't conflict with our use of the separators. If no character was specified in the control panel, the 'itl0' resource contains Char(0). It's OK not to have a thousands separator, but you can't display floating-point numbers without a decimal separator. We don't use GetIntlResource (IUGetIntl), so the outcome of this routine doesn't depend on the font in the current graphics port or the international resources selection flag.

```
PROCEDURE InitializeDefaultNumberSeparators;
VAR   theItl0Handle: Handle;
BEGIN
   theItl0Handle := GetResource('itl0', GetScript(smSystemScript,
                                                  smScriptNumber));
   FailNILResource(theItl0Handle);
   WITH Intl0Hndl(theItl0Handle)^^ DO BEGIN
      IF (decimalPt IN ['0'..'9', Char(0), '-']) OR (thousSep IN
            ['0'..'9', '-']) OR (decimalPt = thousSep) THEN
         FailOSErr(paramErr);
      gDefaultDecimalSeparator := decimalPt;
      gDefaultThousandsSeparator := thousSep;
      END;
END;
```

The FailNILResource, FailNIL, FailOSErr, and FailResError routines check for errors and initiate error handling if necessary; they were originally introduced in MacApp. In this sample code, I don't provide complete error handling, but only call these routines to indicate where a real application would have to be prepared to handle errors.

The procedure LocalizeNumberString takes a string representing a number as it's produced by a nonlocalizable conversion routine and localizes it by adjusting the decimal separator (if there is one) and inserting thousands separators.

```
PROCEDURE LocalizeNumberString(VAR theString: Str255);
VAR   boundary:         Integer;
      separatorString:  String[1];
      minusOffset:      Integer;
BEGIN
   separatorString := ',';
   separatorString[1] := gDefaultThousandsSeparator;
```

First, we find the boundary between the integer and fractional parts. If there's a period, that's the boundary (and we fix the decimal separator right away); otherwise it's the end of the string.

```
   boundary := Pos('.', theString);
   IF boundary <> 0 THEN
      theString[boundary] := gDefaultDecimalSeparator
   ELSE
      boundary := Length(theString) + 1;
```

Second, we insert as many thousands separators as necessary, if the user has specified one. We take into account that we don't want to insert a thousands separator right after a minus sign.

```
   IF gDefaultThousandsSeparator <> Char(0) THEN BEGIN
      IF theString[1] = '-' THEN
         minusOffset := 1
      ELSE
         minusOffset := 0;
      WHILE boundary > 4 + minusOffset DO BEGIN
         theString := Concat(Copy(theString, 1, boundary - 4),
                           separatorString, Copy(theString, boundary - 3,
                           Length(theString) - boundary + 4));
         boundary := boundary - 3;
         END;
      END;
END;
```

And now we finally come to the two routines that an application will call directly to format numbers into strings. The first one is intended for integer numbers, the second one for floating-point numbers.

IntegerToLocalString converts the given integer into a string representation using the thousands separator specified by localization or by the user. It calls NumToString, a Text Utilities routine.

```
PROCEDURE IntegerToLocalString(theNumber: LongInt; VAR theString: Str255);
BEGIN
   NumToString(theNumber, theString);
   LocalizeNumberString(theString);
END;
```

ExtendedToLocalString converts the number into a fixed-point representation using the decimal separator specified by localization or by the user. The number of digits to

NumToString is described in *Inside Macintosh: Text*, page 5-92.•

be used after the decimal separator is specified in decimalDigits. DecForm and Num2Str are defined by SANE.

```
PROCEDURE ExtendedToLocalString(theNumber: Extended; decimalDigits:
                                    Integer; VAR theString: Str255);
VAR   theDecForm: DecForm;
BEGIN
   WITH theDecForm DO BEGIN
      style := fixedDecimal;
      digits := decimalDigits;
      END;
   Num2Str(theDecForm, theNumber, DecStr(theString));
   LocalizeNumberString(theString);
END;
```

That's all there is to the simple case.

NUMBER DISPLAY IN A USER-SPECIFIED FORMAT

Now I'll show you how to use the Text Utilities routines and 'itl4' resources to format numbers according to the user's specification. The idea is that your application comes with a range of predefined format strings, from which the user can pick one. The application might also let users enter their own format strings. Of course, these strings aren't exactly the most user-friendly way to define a number format, so if your application is intended for novice users you should hide them behind a friendlier user interface. Before we dive into the code, let's look at a few obstacles that the Text Utilities routines provide for us and consider how we can work around them.

First of all, the Text Utilities routines expect to work with localized format specifications. They aren't able to take, for example, the standard number format used in the United States and translate it into the standard number format used in Greece, which uses parentheses to indicate negative numbers. This will be a problem if your application doesn't get localized for all regions for which Macintosh system software is localized and if some versions of your application get used in regions for which they aren't localized. To work around this problem, the range of format strings that a given version of the software offers should include all formats commonly used in any of the regions in which this version might be used, and your application should also let users enter their own format strings.

Second, the format strings are interpreted with reference to the characters defined by the number parts table that you pass into the Text Utilities StringToFormatRec routine. You have to be sure to use a number parts table whose characters are compatible with the strings you provide. Currently, only the characters defined by the U.S. 'itl4' resource are documented. To deal with this situation, we'll take advantage of the fact that the U.S. 'itl4' resource is always available and will use its characters as

DecForm and Num2Str are described in the *Apple Numerics Manual*, second edition, pages 26 to 27.•

a stable reference point. We'll define all format strings using the U.S. characters, and use the number parts table in the U.S. 'itl4' resource to interpret them.

Third, if you match a format string against the number parts table of the default 'itl4' resource, you'll have to make sure that your application doesn't break if the user defines a custom number format and the Numbers control panel patches the new decimal and thousands separators into the 'itl4' resource. Therefore, we'll have to undo all changes that the user may have made with the control panel before we can use an 'itl4' resource to interpret our format strings.

Unfortunately, there's no guarantee that other parts of the number parts table won't be modified by the Numbers control panel in the future. This means that a format string that can be converted under the current mechanism may become unconvertible in the future, just as a valid 7.0 format string may no longer be recognized by the 7.1 StringToFormatRec routine. There's not much you can do about this until you know it's happened.

An alternative approach that avoids this problem is to store internal representations of format specifications instead of format strings in the application's resources. The internal representations are created by a separate tool that's run on unmodified U.S. system software during the development process. This approach, however, makes it more difficult for localizers to look at the format strings and to create new ones, and also makes it slightly more difficult to use an additional feature that we'll discuss later, in the "Converting Format Strings" section. I therefore prefer to keep format strings in the application and convert them at run time.

So, to display numbers in a user-specified format, we do the following:

1. Define all format strings using the characters given by the default U.S. 'itl4' resource and documented in *Inside Macintosh: Text*, pages 5-39 to 5-43.
2. Set up two separate number parts tables: one "reference" table that will be used to interpret the predefined format strings and one "user" table that reflects the user's formatting needs.
3. Using the reference number parts table, convert the predefined format strings into the Text Utilities' internal numeric representation.
4. Using the user number parts table, convert the internal numeric representation into format strings that can be displayed to users (to let them select a preferred format, for example).
5. Using the user number parts table, convert a format string that the user has entered into an internal numeric representation.
6. Format numbers using the user number parts table.

Inside Macintosh: Text, pages 5-35 to 5-44, goes into great detail about how to format numbers according to the user's specifications. The approach I describe here differs somewhat from that approach. Instead of assuming that all format strings are localized for the language supported by the default 'itl4' resource, we prepare the application to support format strings from two different sources, the application resources and the user, by using separate number parts tables for them. This is shown in Figure 2, which essentially replaces the upper left portion of the data flow diagram on page 5-37 of *Inside Macintosh: Text*. In all other places where the diagram in *Inside Macintosh: Text* shows a number parts table, we use the user number parts table.

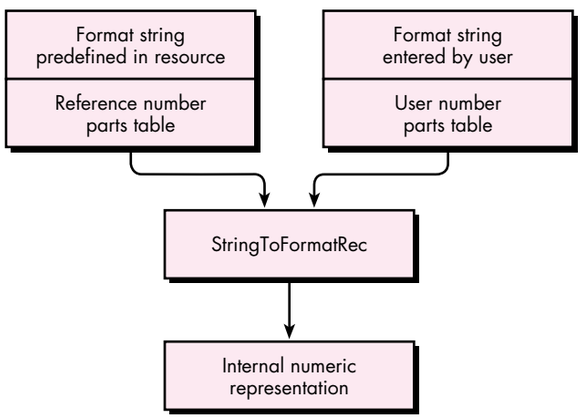


Figure 2
Number Formatting Using Two Different Number Parts Tables

DEFINING FORMAT STRINGS

Which format strings you provide with your application depends on the countries you want to target and the specific needs of your users. Here are some sample strings that you may want to use:

- '###,###.##;-###,###.##;0.##' can be used for floating-point numbers with an absolute value of less than one million, with a thousands separator and the minus sign for negative numbers, and without padding.
- '###,###.##;(###,###.##);0.##' is similar, but with parentheses to represent negative numbers, as is customary in some countries.
- '+^^^;-^^^;^^^^' can be used for integer numbers with an absolute value of less than one thousand, with signs for both positive and negative numbers, and with padding to four places with a space character as may be necessary for alignment.

When you define the format strings, there are a few things you have to watch out for. Most important, the predefined format strings shouldn't include any literal text, as this text is almost guaranteed to be inappropriate for the regions the application isn't localized for. (Note that it may be inappropriate even for the target region; for example, you shouldn't embed currency symbols, because many users deal with foreign currency.)

Also, conversion from U.S. to localized characters really only works for characters that are listed as separate tokens in *Inside Macintosh: Text*, page B-56, not for unquoted characters or other text. For example, parentheses are unquoted characters and don't get converted to the special right-to-left parentheses used by the Arabic and Hebrew script systems. As a result, neither a format string that contains parentheses nor numbers formatted with it display correctly on Arabic, Persian, or Hebrew system software. To avoid problems like this, make sure to test your software on the system software for all countries that you want to target.

Finally, the format string specifies the maximum number of predecimal digits in a formatted number, and the formatting routine will report an error if the number doesn't fit into the format. This means that your program has to ensure that the format strings have enough predecimal digits to accommodate all numbers that may need to be formatted. If the predefined strings you need get too long, you may want to use a simplified version that you can show to users without scaring them.

SETTING UP THE NUMBER PARTS TABLES

Now we'll set up the number parts tables that we'll use. The reference table is based on the U.S. 'itl4' resource, but we'll undo all changes that the user may have made with the control panel. The user table is based on the system's default number parts table and the user's selections in the Numbers or International control panel.

Again, we assume that your application doesn't check for changes in the Script Manager state, and therefore we cache the number parts tables that we need at launch time. The tables are initialized by calling InitializeNumberPartsTables. If the user number parts table is an unmodified U.S. table, only one table is allocated, and both pointers reference this table.

```
VAR   gUserNumberPartsTable:        NumberPartsPtr;
      gReferenceNumberPartsTable:   NumberPartsPtr;
VAR   gSystemVersion:               LongInt;
```

The function GetUserItl4 gets the 'itl4' resource that matches the user's selection in the Numbers or International control panel. This isn't necessarily the default 'itl4' resource.

Number parts tables are described in more detail in *Inside Macintosh: Text,* pages B-55 to B-57.•

```
FUNCTION GetUserItl4:           Handle;
VAR   theItl4Handle:            Handle;
      systemScript:             ScriptCode;
      tableOffset, tableLength: LongInt;
      theItl0Handle:            Handle;
      theResID:                 Integer;
      theResType:               ResType;
      theResName:               Str255;
```

System 7 provides a new routine, GetIntlResourceTable (IUGetItlTable), that returns the number parts table of the default 'itl4' resource. However, the effect of the International control panel on the default 'itl4' resource differs from that of the Numbers control panel, as explained earlier, and we take some extra steps to achieve the behavior that best matches the control panel's behavior. The International control panel selects an 'itl0' resource but doesn't affect the 'itl4' resource. If we continued using the default 'itl4' resource, the user wouldn't see any effect from the control panel selection. To make up for this, we'll try to find an 'itl4' resource that matches the 'itl0' that the user selected, and use it instead of the default 'itl4'. The Numbers control panel, on the other hand, updates the decimal and thousands separators in the 'itl4' resource, and changes in the default 'itl0' resource are limited to the features visible in the control panel. Therefore, the best solution in this case is to use the default 'itl4' resource.

```
BEGIN
   IF gSystemVersion >= $0710 THEN BEGIN
      systemScript := GetEnvirons(smSysScript);
      IUGetItlTable(systemScript, iuNumberPartsTable, theItl4Handle,
                    tableOffset, tableLength);
      FailNILResource(theItl4Handle);
      END
   ELSE BEGIN
```

The workaround used here is to ask the system for the 'itl0' resource and then try to find an 'itl4' resource with a matching number. Some countries, however, have multiple 'itl0' resources and only one 'itl4' resource, or they don't have any 'itl4' resource of their own (they use the U.S. version). To cover these cases, we have to go through an exception table.

```
      theItl0Handle := IUGetIntl(0);
      FailNILResource(theItl0Handle);
      GetResInfo(theItl0Handle, theResID, theResType, theResName);
      theItl4Handle := GetResource('itl4', theResID);
      IF ResError = resNotFound THEN BEGIN
```

The exceptions in system software versions 7.0 and 7.0.1 are as follows:

GetIntlResourceTable (IUGetItlTable) is described in *Inside Macintosh: Text,* pages 6-91 to 6-92.•

- the Netherlands: 'itl0' IDs 5 and 6; 'itl4' only ID 5.
- Czechoslovakia: 'itl0' IDs 30776, 30777, 56, 57; 'itl4' only ID 30776.

Note that 'itl0' 56 occurs in both the Czechoslovakian and Polish versions. For all other cases, we try the U.S. resource.

```
         CASE theResID OF
            6: theResID := 5; { Netherlands }
            30777, 56, 57: theResID := 30776; { Czechoslovakia }
            OTHERWISE theResID := verUS;
            END;
         theItl4Handle := GetResource('itl4', theResID);
         FailNILResource(theItl4Handle);
         END
      ELSE
         FailNILResource(theItl4Handle);
      END;
   GetUserItl4 := theItl4Handle;
END;
```

In the procedure InitializeDefaultNumberSeparators we've verified that the characters the user has specified as decimal and thousands separators don't conflict with the use of these separators for default formatting. Here, we have to take one additional step: the Text Utilities routines for user-specified formatting don't work if the same character is used for different purposes in the number parts table. For instance, a character can't be used both as the decimal separator and to represent digits in a format string. If the routines find a number parts table they don't like, they return the fBadPartsTable result. The procedure CheckDefaultNumberSeparators uses this to check for problems in the user number parts table (which contains the default separators) at application initialization time. In real life, your application should produce a more meaningful message explaining to the user what went wrong and then should quit.

```
PROCEDURE CheckDefaultNumberSeparators(userNumberPartsTable:
                                          NumberPartsPtr);
CONST testString = '0';
VAR   formatRecord:  NumFormatString;
      result:        FormatStatus;
BEGIN
   result := Str2Format(testString, userNumberPartsTable^, formatRecord);
   IF FormatResultType(result) <> fFormatOK THEN
      FailOSErr(paramErr);
END;
```

The procedure ExtractNumberPartsTable is used by the InitializeNumberPartsTables routine to extract a number parts table from an 'itl4' resource.

```
FUNCTION ExtractNumberPartsTable(theItl4Handle: Handle): NumberPartsPtr;
VAR   tableOffset:   LongInt;
      tableLength:   LongInt;
      theTable:      Ptr;
BEGIN
   WITH NItl4Handle(theItl4Handle)^^ DO BEGIN
      tableOffset := defPartsOffset;
      tableLength := defPartsLength;
      END;
   theTable := NewPtr(tableLength);
   FailNIL(theTable);
   LoadResource(theItl4Handle); { Might have been purged since we got
                                  hold of it }
   FailResError;
   BlockMove(Ptr(LongInt(theItl4Handle^) + tableOffset), theTable,
             tableLength);
   ExtractNumberPartsTable := NumberPartsPtr(theTable);
END;
```

The procedure InitializeNumberPartsTables initializes gUserNumberPartsTable and gReferenceNumberPartsTable by copying the tables from the respective 'itl4' resources into nonrelocatable blocks in the heap and cleaning the reference table if necessary.

```
PROCEDURE InitializeNumberPartsTables;
VAR   userItl4, usItl4: Handle;
BEGIN
   userItl4 := GetUserItl4;
   usItl4 := GetResource('itl4', verUS);
   FailNILResource(usItl4);
   gUserNumberPartsTable := ExtractNumberPartsTable(userItl4);
   CheckDefaultNumberSeparators(gUserNumberPartsTable);
```

We check whether the user number parts table is an unmodified U.S. table, so we can use it as the reference table as well.

```
   IF (usItl4 = userItl4) AND ((gSystemVersion < $0710) OR
         ((gDefaultDecimalSeparator = '.') AND
         (gDefaultThousandsSeparator = ','))) THEN
      gReferenceNumberPartsTable := gUserNumberPartsTable
   ELSE BEGIN
```

We need to get the U.S. number parts table and undo any changes that the user may have made with the Numbers control panel.

```
      gReferenceNumberPartsTable := ExtractNumberPartsTable(usItl4);
      gReferenceNumberPartsTable^.data[tokDecPoint].a[1] := '.';
      gReferenceNumberPartsTable^.data[tokThousands].a[1] := ',';
      END;
END;
```

The procedure DisposeNumberPartsTables disposes of the global number parts tables.

```
PROCEDURE DisposeNumberPartsTables;
BEGIN
   IF gReferenceNumberPartsTable <> gUserNumberPartsTable THEN
      DisposPtr(Ptr(gReferenceNumberPartsTable));
   DisposPtr(Ptr(gUserNumberPartsTable));
   gReferenceNumberPartsTable := NIL;
   gUserNumberPartsTable := NIL;
END;
```

CONVERTING FORMAT STRINGS

Now that we have the two number parts tables, we're going to use them to do some conversions. We're going to convert our predefined format strings into the Text Utilities' internal numeric representation, convert this representation into format strings that can be displayed to the user, and convert format strings entered by the user into internal representations.

But first, remember the alternate numerals table mentioned earlier? It's time now to reveal a previously undocumented feature: if a number parts table used for converting a format string to its internal numeric representation contains a character other than "#" as the no-leader format marker, the resulting internal numeric representation will specify using the alternate numerals.

We can use this knowledge to write a wrapper around the StringToFormatRec (Str2Format) routine that temporarily replaces the no-leader format marker, adjusts the format string to use the replacement character as well, calls StringToFormatRec, and reverts the number parts table to its original state. A convenient replacement character is "1," because digits are very unlikely to be used for any other purpose in any version of the number parts table. As the character code for "#" is in a range that isn't used for the bytes of 2-byte characters, we don't have to check for 2-byte characters here.

Here's what the wrapper routine looks like:

StringToFormatRec (Str2Format) is described in *Inside Macintosh: Text,* pages 5-95 to 5-96.•

```
PROCEDURE StringToFormatRecord(formatString: Str255;
                               useAlternateNumerals: Boolean;
                               theNumberPartsTable: NumberPartsPtr;
                               VAR formatRecord: NumFormatString);
VAR   result:  FormatStatus;
      oldChar: WideChar;
      i:       Integer;
BEGIN
   IF useAlternateNumerals THEN BEGIN
      oldChar := theNumberPartsTable^.data[tokNonLeader];
      theNumberPartsTable^.data[tokNonLeader].b := Ord('1');
      FOR i := 1 TO Length(formatString) DO
         IF formatString[i] = '#' THEN
            formatString[i] := '1';
      END;
   result := Str2Format(formatString, theNumberPartsTable^, formatRecord);
   IF useAlternateNumerals THEN
      theNumberPartsTable^.data[tokNonLeader] := oldChar;
   IF FormatResultType(result) <> fFormatOK THEN
      FailOSErr(paramErr);
END;
```

When do you use alternate numerals? First you have to find out whether the user 'itl4' resource you're using supports alternate numerals. You can use the following routine to do this. If it returns TRUE, you should let the user make the final decision whether to use the alternate numerals for output — you can't take for granted that they're always preferred over the ASCII digits. For input, it probably makes sense to accept them without bothering the user first. An exception is the alternate numerals in bidirectional scripts, where the internal representation of the number won't match what the user sees on the screen. You probably shouldn't accept these digits.

```
FUNCTION HasAlternateNumerals(aNumberPartsTable: NumberPartsPtr): Boolean;
BEGIN
   HasAlternateNumerals :=
         aNumberPartsTable^.altNumTable.data[0].b <> Ord('0');
END;
```

Obviously, the decision whether to use alternate numerals has to be made at run time. This is the second reason I recommended storing format strings and not internal representations in the application: with internal representations, you would have to store both versions and select the right one at run time; with format strings, you only store one version and decide at run time how to convert it.

Given this preparation and the two number parts tables, the remaining steps are straightforward. The following routines do no more than call StringToFormatRecord

and FormatRecToString with the appropriate number parts table and check for errors that might occur.

PredefinedStringToFormatRecord converts a predefined format string using the standard U.S. number parts table into an internal numeric representation.

```
PROCEDURE PredefinedStringToFormatRecord(predefinedFormatString: Str255;
                                         useAlternateNumerals: Boolean;
                                         VAR formatRecord: NumFormatString);
BEGIN
   StringToFormatRecord(predefinedFormatString, useAlternateNumerals,
                        gReferenceNumberPartsTable, formatRecord);
END;
```

FormatRecordToUserString converts an internal numeric representation into a format string that can be displayed to the user.

```
PROCEDURE FormatRecordToUserString(formatRecord: NumFormatString;
                                   VAR userFormatString: Str255);
VAR   result:     FormatStatus;
      positions:  TripleInt;
BEGIN
   result := Format2Str(formatRecord, gUserNumberPartsTable^,
                        userFormatString, positions);
   IF FormatResultType(result) <> fFormatOK THEN
      FailOSErr(paramErr);
END;
```

UserStringToFormatRecord converts a format string entered by the user into an internal numeric representation.

```
PROCEDURE UserStringToFormatRecord(userFormatString: Str255;
                                   useAlternateNumerals: Boolean;
                                   VAR formatRecord: NumFormatString);
BEGIN
   StringToFormatRecord(userFormatString, useAlternateNumerals,
                        gUserNumberPartsTable, formatRecord);
END;
```

FORMATTING NUMBERS

After all the preparations, the formatting itself is trivial. FormatNumber formats theNumber into a string, using the internal numeric representation given and the user number parts table.

FormatRecToString (Format2Str) is described in *Inside Macintosh: Text*, pages 5-96 to 5-98.•

```
PROCEDURE FormatNumber(theNumber: Extended;
                       theFormatRecord: NumFormatString;
                       VAR theString: Str255);
VAR   result: FormatStatus;
BEGIN
   result := FormatX2Str(theNumber, theFormatRecord,
                         gUserNumberPartsTable^, theString);
   IF FormatResultType(result) <> fFormatOK THEN
      FailOSErr(paramErr);
END;
```

NUMERIC INPUT

Now let's look at conversions in the opposite direction. When the user enters a number, your application receives a numeric string that it has to convert into a number. This could be quite a difficult task, given that a user may pick a rather arbitrary format (and remember, Macintosh users are generally inclined to do things their own way). Unfortunately, the Toolbox doesn't provide a routine that simply converts an arbitrary numeric string to a number; your application always has to specify the acceptable format.

We can reasonably make some simplifying assumptions: If your application doesn't support output in formats other than ASCII digits, it's probably acceptable to apply the same restriction to the input formats that can be used. And if your application uses only the default format for display, you can also get away with allowing input in this format only, although it would be nicer to accept input in any format the user prefers. If your application supports user-specified number formats, however, it should be prepared to accept input in any currently defined format, and probably in some variations of them, plus the default format.

The Toolbox provides three routines that convert numeric strings into numbers. StringToNum can parse integer numbers but can't deal with anything that goes beyond a sequence of decimal digits that's possibly preceded by a sign. The SANE routine Str2Num can parse floating-point numbers and does detect erroneous input, but it assumes the period as the decimal separator and doesn't support thousands separators. Finally, StringToExtended is supposed to support input of numeric strings in a localized format, but it can deal with only one format at a time and has several other shortcomings that we'll look at later.

We'll take the same approach as with number display and provide separate routines that can deal with the default number format and with user-specified formats.

INTERPRETING NUMBERS IN THE DEFAULT NUMBER FORMAT

This solution restricts user input to something that's very close to the default number format. Then we can simply reverse the "localization" process done for the default

StringToNum and StringToExtended (FormatStr2X) are described in *Inside Macintosh: Text*. See pages 5-93 to 5-94 for StringToNum, and pages 5-98 to 5-99 for StringToExtended.•

Str2Num is described in the *Apple Numerics Manual*, second edition, pages 25 to 26.•

format — that is, strip thousands separators and modify the decimal separator — and call the StringToNum routine or SANE directly to interpret the number.

The procedure UnlocalizeNumberString checks whether theString is a legal number string, strips all thousands separators, and replaces the default decimal separator with a period. As the default number format consists only of ASCII digits, the minus sign, and thousands and decimal separators (which are limited to one byte by the 'itl0' resource), we don't have to worry about 2-byte characters here if we walk the string and check each character (we would have to worry if we used a string search routine).

```
PROCEDURE UnlocalizeNumberString(VAR theString: Str255;
                                 allowDecimal: Boolean);
VAR   delta:      Integer;
      i:          Integer;
      theChar:    Char;
BEGIN
   delta := 0;
   FOR i := 1 TO Length(theString) DO BEGIN
      theChar := theString[i];
      IF (theChar >= '0') & (theChar <= '9') THEN
         theString[i - delta] := theChar
      ELSE IF (theChar = '-') & (i = 1) THEN
         theString[i - delta] := theChar
      ELSE IF theChar = gDefaultThousandsSeparator THEN
         delta := delta + 1
      ELSE IF theChar = gDefaultDecimalSeparator THEN BEGIN
         IF allowDecimal THEN BEGIN
            allowDecimal := FALSE; { one is enough }
            theString[i - delta] := '.';
            END
         ELSE
            FailOSErr(paramErr)
         END
      ELSE
         FailOSErr(paramErr);
      END;
   theString[0] := Char(Length(theString) - delta);
   IF Length(theString) = 0 THEN
      FailOSErr(paramErr);
END;
```

And here we finally come to the two routines that your application will call directly to interpret numeric strings. The first one is intended for integer numbers, the other one for floating-point numbers.

LocalStringToInteger converts to an integer a numeric string that we're hoping represents an integer. The string may contain localized thousands separators.

```
PROCEDURE LocalStringToInteger(theString: Str255; VAR theNumber: LongInt);
BEGIN
   UnlocalizeNumberString(theString, FALSE);
   StringToNum(theString, theNumber);
END;
```

LocalStringToExtended converts to an equivalent floating-point number a numeric string that we're hoping represents a fixed-point number. The string may contain localized thousands and decimal separators.

```
PROCEDURE LocalStringToExtended(theString: Str255;
                                VAR theNumber: Extended);
BEGIN
   UnlocalizeNumberString(theString, TRUE);
   theNumber := Str2Num(theString);
END;
```

INTERPRETING NUMBERS IN USER-SPECIFIED NUMBER FORMATS

Now we can benefit from some of the work that we did to display a number in a user-specified number format: we use the same number parts tables and the same format conversion routines. However, because we can't really constrain users to any given format for input, we have to allow them at least to use any currently defined format, including the default format. To do this, your application should try the default format and all currently defined format strings until it finds one for which the conversion is successful.

The core of this conversion is the StringToExtended (FormatStr2X) routine, and it helps to understand how this routine works. You get the most reliable results from StringToExtended if users enter numbers exactly in one of the three possible formats given by a format string (for positive and negative numbers and 0). In this case it returns fFormatOK. If the input string doesn't conform to any of the three formats, StringToExtended tries to guess: it replaces the default decimal separator in the string with a period, moves the minus sign to the beginning of the string, and strips some other punctuation. If the resulting string can be interpreted by SANE, StringToExtended returns fBestGuess and the number found by SANE; otherwise StringToExtended returns one of several other values that indicate why the string cannot be interpreted.

You can control how liberal your application is by either allowing fBestGuess as a result of StringToExtended or treating it as an error. If you don't allow fBestGuess, the input string can deviate from the specified format in only the following two ways:

Result values for StringToExtended are described in *Inside Macintosh: Text,* page 5-38.•

- If you specify parentheses to indicate negative numbers, StringToExtended recognizes parenthesized numbers as negative but still also accepts numbers with a minus sign. This helps in many countries where negative numbers can be written with either a minus sign or parentheses.
- If you set useAlternateNumerals to TRUE when you converted your format string, StringToExtended accepts ASCII digits in addition to the alternate numerals.

If you allow fBestGuess, the following deviations (and more) are also allowed:

- The minus sign for negative numbers can occur anywhere in the number. That may be nice for some people who prefer to write the minus at the end, but interpreting "12–3" as –123 doesn't make sense.
- Thousands separators can be missing or in the wrong place.
- A fractional part can be present while the format string doesn't provide for it.
- A decimal separator and fractional part specified in the format string can be missing.
- More predecimal digits than specified by the format can be present.

Because the guessing process ignores some of the information entered by the user, it may behave inconsistently with the behavior shown when the format is OK. For example, if your format string uses parentheses to indicate negative numbers and also specifies thousands separators, "(123)" will be read as -123 (with fFormatOK), but "(1234)" will be read as 1234 (with fBestGuess). The missing thousands separator causes StringToExtended to go into the guessing pass, and there the parentheses are stripped instead of being interpreted as the indicator for "negative."

This situation is somewhat unfortunate. On one hand, checking for fFormatOK alone doesn't give users much freedom to type numbers in their preferred format — for instance, where a fractional part is specified, they must at least enter the decimal separator. On the other hand, allowing fBestGuess may mean that some pretty bizarre strings are accepted and may lead to inconsistent interpretation of parentheses. It seems that for integer strings allowing fBestGuess is never a good solution, while for real values it might be OK as long as you don't support expressing negative numbers with parentheses. To play it safe, I don't allow fBestGuess in the code that follows.

To provide the appropriate handling and return the desired type of number, here are two different routines that interpret numeric strings representing floating-point and

integer numbers. They both return Booleans to tell the caller whether theString was successfully converted.

InterpretExtended interprets a numeric string and converts it to an Extended number.

```
FUNCTION InterpretExtended(theString: Str255;
                           theFormatRecord: NumFormatString;
                           VAR theNumber: Extended): Boolean;
VAR   result: FormatStatus;
BEGIN
   result := FormatStr2X(theString, theFormatRecord,
                           gUserNumberPartsTable^, theNumber);
   InterpretExtended := FormatResultType(result) = fFormatOK;
END;
```

InterpretInteger interprets a numeric string that (we hope) represents an integer and converts it to the equivalent LongInt.

```
FUNCTION InterpretInteger(theString: Str255;
                           theFormatRecord: NumFormatString;
                           VAR theNumber: LongInt): Boolean;
VAR   result:        FormatStatus;
      theExtended:   Extended;
CONST
      minLongInt = -2147483648;
      maxLongInt = 2147483647;
BEGIN
   result := FormatStr2X(theString, theFormatRecord,
                           gUserNumberPartsTable^, theExtended);
   IF (FormatResultType(result) = fFormatOK) & (theExtended >= minLongInt)
                                 & (theExtended <= maxLongint) THEN BEGIN
      theNumber := Num2LongInt(theExtended);
      InterpretInteger := TRUE;
      END
   ELSE
      InterpretInteger := FALSE;
END;
```

TESTING THE CODE

The Numbers Test application on this issue's CD tests all the routines that we've seen so far. It's a simple, text-based application that spits out a few numbers in the default format, converts a few format strings and displays them in the localized version, and then runs a test that reads in numbers and formats them according to format strings.

Num2LongInt is described in the *Apple Numerics Manual,* second edition, page 22.•

In a real program, you would of course get the format strings and the messages from resources. In case you prefer to play with an MPW tool, you can use the BuildNumbers script on the CD to build a tool that's equivalent to the application.

One final warning in case you don't check out the source code: Before using the routines described in this article, your application has to check that it runs on system software version 7.0 or higher, because the section "Setting Up the Number Parts Tables" makes some assumptions about this.

OFF YOU GO

If you compare the user expectations described at the beginning of this article and what we've actually achieved with the Toolbox, you'll find that we've been only partially successful. The code handles the default number format and some user-specified customizations, but it's still limited to decimal numbers expressed in ASCII digits and to additional characters supported by the 'itl4' resource.

You may consider adding code of your own to make your number handling more flexible. There's an obvious need for support for nondecimal numbers. Many specialized applications already support Roman numerals or hexadecimal numbers; your application could similarly support traditional Chinese numbers or decimal numbers using Chinese digits.

When you add your own number support, you should always design it as an extension to the facilities provided by system software, not as a replacement. The Macintosh is being localized into ever more languages to reach ever more customers, and you'll be more successful if your application is at least as good as system software worldwide.

REFERENCES

- *Apple Numerics Manual,* 2nd ed. (Addison-Wesley, 1988).
- *Inside Macintosh: Text* (Addison-Wesley, 1993). See Chapter 1, "Introduction to Text on the Macintosh"; Chapter 5, "Text Utilities"; Chapter 6, "Script Manager"; and Appendix B, "International Resources."

THANKS TO OUR TECHNICAL REVIEWERS

Jeannette Cheng, Peter Edberg, Bryan K. ("Beaker") Ressler, Kenny Tung•

VIEW FROM THE LEDGE

TAO JONES

Dear Tao,

I'm nearing the end of my rope and have become desperate enough that I figure even a letter to you couldn't hurt. Over the last couple of years I've found myself working longer and longer hours. I pointed this out to my boss and she said that there's no way the company could afford to hire more people, and that I was going to have to learn how to "work smarter." Just what the heck is that supposed to mean?

P. O.

Dear P.,

In order to work smarter you have to do two things: first figure out who is smarter than you, and then copy their actions and claim them as your own. This is a problem that's easily solved if you look at history.

Unless your name is Stephen Hawking, it's pretty safe to say that Albert Einstein was smarter than you. Let's see how he handled a similar situation.

Early in his career Mr. Einstein was working in a patent office, but he was having trouble keeping up with the ever increasing workload. Upon asking his boss for advice, he too was told to "work smarter." He was also advised that he'd be a little more presentable around the office if he wore a hairnet while he slept.

Very disturbed, he went home and contemplated these words. The story is told that after several hours of soul searching and a couple of Fuzzy Navels he was able to get in touch with his Inner Physicist Self: he quit his job, came up with the General Theory of Relativity, and as a result started a career as a professional smart guy.

Taking the cue from Al, I'd say the answer's simple: quit your job and never go back to anything even remotely related to it. True, it's not likely that you'll win the Nobel Prize for physics, but I'm certain that at the very least you'll find it much easier to balance your checkbook (a task that is quite a bit simpler when you don't have to worry about income).

As for wearing the hairnet, I'd say that's optional. On the off-chance you *do* become famous, you won't look nearly as cool when they make a poster of you if your hair is neat and tidy — something always worth considering in any situation.

Dear Tao,

I keep a pet hamster at work, and the guy across the hall has a rat. We're thinking of getting one of those cages that link together so that our pets can visit each other, but before we do I have a question: would it be possible for them to have babies? I think it'd be really cool and break up the monotony of the day.

Sign me,

Dr. Doverylittle

Dear Doctor,

It sounds like you may have not been paying very close attention to your biology classes for, say, your last ten years of school. The short answer is no, you can't cross-breed rats and hamsters. However, you'll get a brief but downright eye-opening demonstration of what food chains are all about if you decide to take the leap and give your pets a formal social introduction.

TAO JONES, in a high school English class assignment years ago, was asked to describe his ultimate fantasy. Although the exact details are now forgotten, it had something to do with huge castles, large bicycle tracks, and "The Gong Show." When he discovered that the rest of the class had described things like "a long weekend with Farrah Fawcett," he immediately started working on an all-encompassing philosophy to explain the rest of humanity. It's taken years to perfect, and today this philosophy is commonly referred to as his "bad attitude."•

Tao Index: Do not trust those who have their jeans pressed.•

Speaking of rodents, it's worth noting that they can be of tremendous help in your company's advertising efforts. Let's say you have a list of possible slogans to use but are unsure which is best. Simply tape all the slogans you're considering on the side of your rodent tank and then watch carefully. The one your little beast walks toward will be the natural winner. Remember, advertising works on the most primitive portions of the brain, so a rat is as good as an ad exec for the task, and you get the added benefit of Purina Rat Chow being quite a bit cheaper than the cost of a two-martini lunch.

Can't even come up with a slogan? No problem. You can use single words just as easily and then string them together. It might take a few trials until you get something that makes sense, but keep at it. I tried this using random words cut out of the newspaper and came up with the ultra-highbrow "Price last, anybody's near Russia." It's got everything you need: concern for the customer *and* faux concern for other countries. Even today I'll bet you could sell a few Lisas using that heady doublespeak.

Dear Tao,

I had a conversation with my boss the other day in which he accused me of "gross incompetence" merely because I was personally responsible for the failure of our company's last three products. I could use your advice on how to proceed.

Spud Boy

Dear Spud,

If what you said is true, your boss is right: you are incompetent. Don't let that bother you; competency is only one of several factors related to your job, and in some ways your future has never looked brighter. However, you should take action if you're interested in staying where you are.

The key here is to slyly change your company's employment guidelines. In the 1990s nearly all businesses of any size have a bylaw that says something like, "Sproutbud Software has a policy of equal opportunity hiring and we will not discriminate with regard to an employee's ethnic origin, religious beliefs, age, sex, sexual preference, marital status, or even if they wear brown shoes with a blue suit."

Obviously the phrase that you want added is ". . . will not discriminate with regard to ability," but that's going to be hard to get on the first pass. What you'll need to do is get very militant about something that seems relatively trivial. The next time you have a large meeting, hint that you have reason to believe that your company discriminates against people who like sherbet. Once they deny it (as they undoubtedly will), say "OK, maybe you're right, but I won't be comfortable until our employment guidelines are changed to reflect this."

Next, insist there is discrimination against people who like the color magenta; then, those with a penchant for keeping more than a dollar's worth of change in their pockets. Eventually people will get so sick of you that they'll just say "Yeah, whatever! Just fill out whatever you like. Anything to get you to shut up!"

Be careful. Once you have this power you'll feel like flaunting it by saying something like ". . . will not discriminate against people whose last names end in *y*." It sounds catchy, but it's a sure tip-off to your plans.

RECOMMENDED READING AND LISTENING

- *The Airline Passenger's Guerrilla Handbook* by George Albert Brown. Get the lowest airfares *and* learn how to open that bag of peanuts.
- *Shakespeare's Insults* compiled by Wayne F. Hill and Cynthia J. Öttchen. Gives you the weapons you need in case you're ever called a "flinty Tartar."
- *John Lee Hooker, The Ultimate Collection,* Rhino Records. It's time to get blue, irrespective of your day-to-day color.

Used chewing gum? Place it here.•

Tao needs questions or he may be forced to do his real day job. Please send your queries regarding the social and political aspects of office survival to AppleLink DEVELOP. If your question is published, he'll reward you with an incredibly cheap, yet heartwarmingly collectible, gift.•

MACINTOSH Q & A

Q *I want to write an application that can open incoming PowerTalk letters automatically. How can my PowerTalk-savvy application access the letters stored in the in-tray?*

A Currently, there are only two ways of accessing in-tray letters from your application. The most common method is to receive an open document ('odoc') Apple event when a letter is opened from the Finder. In addition, calling SMPGetNextLetter allows you to open letters from your application. It isn't possible, however, for your application to access the in-tray from Standard File or to automatically open arbitrary letters without user intervention.

Q *When I open a letter in my PowerTalk-savvy application, I can obtain FSSpec records referencing folders or files enclosed in the letter, but the files don't appear to be copied to my hard drive. Where do these files reside on my system?*

A PowerTalk maintains an external file system for enclosures of letters. If you closely examine the FSSpec records returned to you, you'll see that the enclosed files and folders reside on this external file system, named "Mail Enclosures." This is a read-only file system, and the enclosures are available only while the letter is open. Therefore, these FSSpecs should be used only to copy the enclosed items to your local disk, and all references to them should be discarded by the time you call SMPDisposeMailer.

Q *When I call FrontWindow from within a VBL task, my system occasionally freezes at this call. Is there any chance that it moves memory?*

A No, but FrontWindow is not reentrant. It's been patched out since MultiFinder, and ought to be on the list of routines that shouldn't be called at interrupt time. You'll have to come up with another method of getting the front window from your VBL task. You may want to keep a shadow copy of your application's window list in your application global area where your VBL task can get to it.

Q *We want a general extension for all PAP-capable printer drivers, to allow for user authentication at print time. It would conduct a user authentication dialog with a spooler on the order of the one described in Chapter 14, "Print Spooling Architecture," of Inside AppleTalk, Second Edition. Is it possible to do this in the QuickDraw GX printing architecture? Would we do anything different for PostScript-savvy print spoolers to insert the right document-structuring comments?*

A For spoolers that are PostScript-savvy, you can override the message GXPostScriptDoDocumentHeader from a printing extension and insert your password information in the PostScript header at that point, including password, user name, and so on. When your spooler receives the job, it can

These answers are supplied by the technical gurus in Apple's Developer Support Center. Special thanks to Sonya Andreae, Mark Baumwell, Brian Bechtel, Chris Berarducci, Matt Deatherage, Tim Dierks, Steve Falkenburg, Nitin Ganatra, Bill Guschwan, Dave Hersey, Jim Luther, Joseph Maurer, Kevin Mellander, Martin Minow, Eric Mueller, Ed Navarrete, Mike Neil, Guillermo Ortiz, Jeroen Schalk, Brigham Stevens, Dan Strnad, and John Wang for the material in this Q & A column. Extra special thanks and a fond farewell to Rilla Reynolds, who took a thankless job and made it work for all of us.

check this data and see if it denotes a valid user. By overriding this message you'll work with any PostScript driver, as long as your spooler supports it. (You're actually talking only to your spooler, not to the different drivers.)

For the authentication, you'll need to override GXOpenConnection to get the user's name and password, since that's when you actually try to connect to the device or spooler. You could use cool alerts (status dialogs) to get the information from the user at that time. Be sure not to put up a "Type in your password, please" dialog except at device communication time. At other times, the user won't be trying to connect to the device and shouldn't need to provide authentication.

Q *We plan to convert all our QuickDraw objects to QuickDraw GX objects when we print, then use a QuickDraw GX printing loop to print our objects. We print bitmaps at the printer's maximum resolution, through PrGeneral. How do we print bitmaps at the printer's maximum resolution in QuickDraw GX? How do we find the printer's maximum resolution?*

A To print bitmaps at the printer's maximum resolution in QuickDraw GX, create the bitmaps at the desired resolution (like 300 dpi) and then put a transform on the shape to scale it by 72 divided by the bitmap's resolution. This method makes the bitmap show up with the correct dimensions, whether it's drawn on a 72-dpi screen or a 300-dpi printer. And, on the 300-dpi printer, the bitmap will not be scaled at all — it will render at 300 dpi. This works because, before drawing, the shape's scaling is multiplied by the device's resolution divided by 72, to convert the shape to device resolution.

To find a printer's maximum resolution, use the calls to get the information from the printer's view device list (get the printer from the job via GXGetJobPrinter), pick the highest-resolution printing device, and voilà!

Q *My application won't print to a StyleWriter II although it works fine on all other printers, including the original StyleWriter. Can you give me any suggestions? What's different about the StyleWriter II?*

A The StyleWriter II driver is a member of the "GrayShare" driver family, with new features such as support for grayscale printing (if Color QuickDraw is available) and printer sharing over the network. Its internal architecture is very different from previous printer drivers. In spite of thorough compatibility testing, some problems have shown up since the first release. In many cases, the driver revealed weaknesses in the applications themselves; for some other problems, a solution will be incorporated in the next release of the driver. Here are some identified problem areas with GrayShare drivers (note that these don't

necessarily represent bugs in the drivers, but are differences in the system's configuration at print time):

- GrayShare drivers handle memory differently. For example, if an application has a dereferenced handle to an unlocked block while doing a CopyBits into the printing port, this may work fine on other printer drivers, but the block is likely to move with a GrayShare driver and the results are unpredictable.
- GrayShare drivers maintain their own A5 world internally. If an application installs a growZone proc and forgets to set up its own A5 in the growZone proc — ignoring the Macintosh Technical Note "Register A5 Within GrowZone Functions" (Memory 1) — the growZone proc may get called with a GrayShare driver's A5, which obviously is bad for the survival of both the application and the printer driver.
- GrayShare drivers call the pIdleProc (in the job subrecord of the print record) more often than other printer drivers; in particular, it may be called during execution of PrOpenDoc. If an application reloads a previously used print record containing an old (now invalid) pointer to a pIdleProc and doesn't update the pIdleProc field before calling PrOpenDoc, disaster is very likely. Note that Macintosh Technical Note "pIdle Proc (or how to let users know what's going on during print time...)" (Printing 22) recommends installing your pIdleProc before PrOpenDoc.
- Like most other QuickDraw printer drivers, GrayShare drivers use algorithms built into QuickDraw for rasterizing picture elements like ovals and arcs, but not necessarily with the same sequence of coordinate transformations. Because of the 360 dpi resolution, internal computations with the transformed coordinates may hit the 16-bit integer limitation of QuickDraw and reveal bugs in the old QuickDraw routines that have remained hidden until now.
- Some GrayShare drivers (like the StyleWriter II driver, version 1.0) contain STR# resources with positive ID numbers that may conflict with STR# resources in an application. This will be fixed in the next release of the drivers.
- Under certain circumstances, GrayShare drivers seem to have trouble with the PmForeColor call. This is under investigation and will be fixed.

Q *We've been manually writing 'PAPA', 'STR ', and 'alis' resources in the System file and in the LaserWriter driver to change printers without using the Chooser. This method sometimes causes errors with LaserWriter 8. What do we need to do?*

A LaserWriter 8 needs to know more about the printer than its AppleTalk name — it also has to have a PostScript printer description (PPD) file for that printer, parsed and ready to be used. Since there's so little memory available in

applications like the Finder during printing, the parsing is done at Chooser time, not at print time.

Apple has *always* said "We can't guarantee that you can change printers behind the Chooser's back," and with LaserWriter 8 this is true. If the driver has parsed a PPD file and has it ready, things should work OK, but everything must have been manually set up by choosing that printer ahead of time. If you set up a LaserWriter IINTX printer with the correct PPD file, choose a LaserWriter IIf, and then choose another printer driver, you could probably programmatically switch back to the LaserWriter IINTX, but the driver will use the LaserWriter IIf PPD file with it, which might or might not produce the right behavior. Designing the driver to be switchable by other applications simply wasn't a priority of the Adobe/Apple development team.

As long as you try to switch to a printer that uses the same PPD file that the driver last parsed (meaning the PPD associated with the last printer selected in the Chooser), there shouldn't be any more problems than there were before.

Q *I want to create a QuickDraw GX font that will calculate check digits as the user types the numbers in. For example, if you type "123458723" the check digit would be 5; if you type "098732734" the check digit would be 7; and so on. The formula is check digit = sum of nine numbers MOD 10; the check digit is appended as the number's tenth digit. Is there a way to do this using the glyph modification properties in QuickDraw GX? I know it can be done by creating an entry in the 'mort' table for all unique possibilities, but there are 9^9 possibilities, which comes to 387,420,489. Can I enter a formula instead of creating tables? If so, how?*

A It's a fascinating idea, but it's not something the 'mort' tables can handle. These tables are basically state machines; when they detect a particular state, they take some action based on entries in the table. There's no way to add a formula or code in any language to these tables.

All the tables can do is take a series of glyphs and replace them with a different glyph if a given feature is enabled by line layout. For example, you can change the two letters "f"and "i" into the corresponding ligature"fi" (one glyph). If you wanted to do this for check digits, you'd have to have an entry in the 'mort' table for each glyph combination you wanted substituted. For 300000000, you'd have to have one entry that substituted 3000000003, and the substitution has to be one glyph. Since order is significant, that means you'd have to have one billion entries in the 'mort' table and one billion glyphs representing all the entries with their check digits added. That's where we have to stop, because a font can't contain more than 65,536 separate glyphs. It's a really neat idea, but it won't work.

For more information on LaserWriter 8,
see this issue's "Print Hints" column.•

Q *I've selected AppleShare volumes to mount at system startup by checking the volumes in the Chooser list. If I'm on a nonextended network and I call an extended network via AppleTalk Remote Access and log into a remote server via the Chooser and AppleShare, an error alert will say "The AppleShare Prep file needed some minor repairs. Some AppleShare startup information may be lost." All the information about my local nonextended network will be cleared out of the AppleShare Prep file. So I lose all my log-in IDs and passwords for my local servers. The same thing happens going back the other way (extended to nonextended). Why is this happening?*

A There are several problems you can run into when you connect two networks (which is what you're doing when you use AppleTalk Remote Access when you're already connected to a network). The problems are usually the result of duplicate names or duplicate node numbers.

The "boot mount list" (BML) kept in the AppleShare Prep file stores the location of volumes that you want mounted at boot time. Part of that location is the zone name. If you create BML entries when you aren't on an extended network — that is, when you have no zones — the zone name stored in the BML is "*" (AppleTalk's shorthand for "this zone"); otherwise, the zone name of the server is stored in the BML.

The boot mounting code checks the validity of the BML when the system starts up, and the AppleShare Chooser package checks the validity of the BML when it's opened. If there are no zones, entries with zone names other than "*" are cleared and the alert "The AppleShare Prep file needed some minor repairs. Some AppleShare startup information may be lost" is displayed because those entries aren't valid. If there are zones, entries with zone names of "*" are cleared and the alert is displayed because the "*" zone name isn't a reliable way to save the zone location of a server on an extended network. The "*" zone isn't reliable for storing the zone name because a workstation can easily be moved from zone to zone, keeping the same NBP object and NBP type names. This is especially true with AppleTalk Phase 2, which supports multiple zones on a single network (for example, multiple zones on the same piece of Ethernet cable).

The workaround for boot-mounting volumes is to create alias files to the file servers you want to mount at boot time and then drop those alias files into the Startup Items folder inside your System Folder. The only drawback to this is that aliases don't save the user's password. If you need boot-mounted volumes without the password dialog, you'll have to use guest access.

Q *What is a Macintosh IIvm? One of the System Enabler files defines the computer in gestaltMachineType 45. Is this just another name for the Performa 600? The Macintosh IIvx Developer Note says that the Performa 600 returns type 45.*

A Apple had planned to release a model called the Macintosh IIvm but consumer testing showed that users thought "vm" was an abbreviation for "virtual memory." This was about the same time Apple was about to introduce the Performa line. So, to avoid confusion, the model became the Performa 600. There are, therefore, three models in the "v" series: the Macintosh IIvi, Performa 600 (Macintosh IIvm), and Macintosh IIvx. As you can see, using the nonreleased "Macintosh IIvm" designation confuses people, so try to avoid it.

Q *What do those numbers at the end of System Enabler file names mean?*

A If you hold down a certain modifier key combination while opening your System Folder, the System 7.1 Finder will reward you with a special message from Apple's Advanced Technology Group — if and only if the current tick count (as returned by TickCount) divided by the number at the end of the machine's enabler file name (in decimal) is exactly equal to the model number of the Macintosh for which the enabler is intended.

Legal restrictions prevent us from revealing the message itself, but enterprising techno-nerds may attempt to find it with these instructions.

Q *Which variables does the stack sniffer VBL task look at to determine that the stack has crossed over into the heap? I create stacks for my own subtasks in the heap. Will modifying those variables affect anything else besides the stack sniffer? What's the correct process to defeat the stack sniffer task (leaving it installed) or remove the task?*

A Disabling the stack sniffer is reasonably simple — storing four bytes of $00 in the low-memory global StkLowPt ($110) will turn the sniffer off. However, when using your own internal stack, be sure *not* to call any Toolbox routines, because many of them rely on the stack for temporary storage, which will screw things up if you've played with the value of register A7.

When you're using your own stack within your heap, you should definitely save the values in StkLowPt and A7 before changing them, so that you can reset their values before and after any Toolbox calls.

Q *How can I tell whether a picture is QuickTime-compressed?*

A The key to your question is "sit in the bottlenecks." If the picture contains any QuickTime-compressed images, the images will need to pass through the StdPix bottleneck. This is a new graphics routine introduced with QuickTime. Unlike standard QuickDraw images, which only call StdBits, QuickTime-compressed images need to be decompressed first in the StdPix routine. Then QuickDraw uses StdBits to render the decompressed image. So swap out the

QuickDraw bottlenecks and put some code in the StdPix routine. If it's called when you call DrawPicture, you know you have a compressed picture. To determine the type of compression, you can access the image description using GetCompressedPixMapInfo. The cType field of the ImageDescription record will give you the codec type.

See the CollectPictColors snippet on this issue's CD and "Inside QuickTime and Component-Based Managers" in *develop* Issue 13, specifically pages 46 and 47, for more information on swapping out the bottlenecks.

Q *Is there a way to embed a QuickTime movie into a Macintosh file containing non-QuickTime stuff and get the Movie Toolbox to play the movie back correctly? If so, can we pass the same movie handle to QuickTime for Windows and get it to play back the same data from the same file?*

A To add QuickTime movie data to non-QuickTime files, just store the movie data in the file using FlattenMovieData with the flattenAddMovieToDataFork flag. Since FlattenMovieData will simply append to a data fork of a file, you can pass it any data file and it will append the movie data to that file. QuickTime doesn't care what's stored before or after the movie data, as long as you don't reposition the movie data within the data file. If you do, the movie references will be incorrect since they aren't updated when you edit the file. The returned movie (from FlattenMovieData) will properly resolve to that data file. You can then save this movie in the data fork with PutMovieIntoDataFork or in the resource fork with AddMovieResource. If the movie is saved in the data fork, it can be retrieved by both QuickTime and QuickTime for Windows with NewMovieFromDataFork.

You can, in fact, store multiple movies simply by calling FlattenMovieData and PutMovieIntoDataFork several times on the same file. Each FlattenMovieData call appends new data, assuming the createMovieFileDataCurFile flag isn't set.

Q *Is there a way that the action filter procedure of a QuickTime movie controller component can have a user reference field so that I can know which movie "object" the movie controller refers to? There are local variables associated with a particular movie that I would like to access from the action filter procedure; currently there's no way to reference back to the variables in my program except through globals.*

A This was a difficult task under QuickTime 1.0, requiring you to stuff some sort of pointer in the movie user data fields. The good news is that in response to developer requests, starting with version 1.5, QuickTime includes a new call in the Movie Controller suite allowing you to pass and receive a long value when you set up your filter procedure, as follows:

```
pascal ComponentResult MCSetActionFilterWithRefCon(MovieController
                        mc, MCActionFilterWithRefCon
                        myUserPlayerFilter, long refCon) =
                        {0x2F3C,0x8,0x2D,0x7000,0xA82A};
```

The procedure is of the form

```
pascal Boolean userPlayerFilter(MovieController mc, short action,
                                void *params, long refCon);
```

The procedure returns true if it handles the action, false if not. The action parameter identifies the action to be executed; params is the set of potential parameters that go with the action; and refCon is any long value passed to MCSetActionFilterWithRefCon.

Q *We want to add temporal compression for our long movies with similar sequential frames. How do I use the Compression Manager routines (CompressionSequenceBegin and CompressSequenceFrame, for instance) in conjunction with the normal movie-making routines (such as CompressImage)?*

A Sequence compression is useful for temporal compression, with processes like animation. Sequence compression works by providing *one* description handle for a series of frames, whereas CompressImage may use a description handle for each image. Thus, functions such as CompressImage are normally used separately from the sequence calls. It's a bit confusing; the Movie Construction FD sample on the QuickTime CD should help clarify how to use the calls.

Sequence compression performs similarly to creating a movie with CompressImage, but the major difference is the specification of key frames. (Key frames are frames against which all the following frames are differenced.) CompressSequenceFrame returns a similarity value, which tells you whether the frame is a key frame (0 means key frame). Based on this value, you can tell AddMediaSample the type of frame it is. Here's some pseudo code:

```
err = CompressSequenceFrame(seqID, ..., similarity, nil);
err = AddMediaSample(gMedia, ..., similarity ? mediaSampleNotSync :
      0, &sampTime);
```

Q *When we try to digitize frames (grabbed with QuickTime) into an off-screen pixel map, our VDGrabOneFrame call crashes. How would you suggest we do this?*

A You need to check whether the 'vdig' resource supports digitizing off-screen by calling the PreflightDestination routine. If the PreflightDestination call with an off-screen destination fails, you need to digitize to a window on the digitizing

device and then copy the image (using CopyBits or your own algorithm for speed) from the window to your off-screen pixel map. Some 'vdig' resources don't support digitizing directly to off-screen pixel maps because their hardware does the digitizing asynchronously. You should always preflight your destination before setting it with SetPlayThruDestination.

Q *Why doesn't the QuickDraw GX LaserWriter driver have a 'pdip' resource? Isn't it required?*

A A 'pdip' resource isn't required; if it isn't present in the driver, the default (LaserWriter Plus) preferences are used. Those preferences are currently as listed below. They're subject to change, so don't depend on them. If you need a specific value for any of these preferences, just include your own 'pdip' resource.

language level	1
device color space	graySpace
device color profile	nil
render options	noOptions
path limit	1496
gsave limit	1
operand stack limit	500
font type	type1Stream + type3Stream
printer VM	200K

Q *Inside Macintosh: Macintosh Toolbox Essentials recommends calling CloseDialog instead of DisposeDialog when allocating memory for a dialog record manually instead of letting the Toolbox do it. But doing so causes a memory leak, because GetNewDialog copies the DITL resource in memory. The DITL copy isn't released by CloseDialog and isn't purgeable, even if the original DITL was purgeable. What's the official method of completely getting rid of a dialog whose storage you've allocated by hand?*

A CloseDialog was intended to mirror NewDialog; it allows you to close a dialog that you provided the storage for, including the item list. Page 6-119 of *Inside Macintosh: Macintosh Toolbox Essentials* states that the item list is specifically not disposed of by CloseDialog, so it's acting as documented. It does this so that it won't dispose of a dialog item list you might be planning to use again. If you do want to dispose of the item list, just do so after calling CloseDialog.

Q *As a MacApp developer, am I supposed to be using the .h files in the MPW:Interfaces folder, or the ones in the {MacApp}CPlusIncludes folder?*

A As a default, MacApp 3.0 first searches through the {MacApp}CPlusIncludes folder for header files specified in your source, so we suggest using the

CPlusIncludes headers. The path to these include files is defined in the {MacApp}Startup Items:Startup folder. MacApp 3.1 uses the universal interfaces for both 680x0 and PowerPC development. It no longer uses its own custom headers in the {MacApp}CPlusIncludes folder. MacApp 3.1 searches MPW's {CIncludes} folder for its headers.

Q *When starting a new MacApp program, I get the message "Couldn't create new document because an internal component is missing. Please contact the developer." How do I find out which component is missing?*

A When building an application, the linker strips out any unused code from the final application. The problem is that it determines which code is to be stripped out by finding all objects that are constructed with the **new** operator. Because objects derived from TView might instead be instantiated through calls to TViewServer methods, the linker thinks that calls to your derived TView objects aren't used, so those objects are stripped from the build. To circumvent this, you have to fool the linker into not stripping out this code. MacApp defines a macro that makes it easy for you to trick the linker. Place this line of code in your implementation of TSomeApplication::ISomeApplication:

```
macroDontDeadStrip(TSomeView);
```

Do this for any subclass of TView defined in your program. Good examples of the use of this routine can be found throughout the MacApp C++ code.

MacApp takes care of this for you for any TView subclasses defined by MacApp, provided you call InitUDialogs, InitUTEView, and so on in your main routine. The exact set of routines you have to call depends on the TView classes you use in your application. All of these routine names begin with "InitU."

MacApp uses two alerts to indicate that you're missing components. The first one ends with "because an internal component is missing. Please contact the developer." The second one is identical except that it's preceded by the class name of the missing component. In the former case it's very likely you forgot one of the "InitU" calls; in the latter case you're very likely missing a macroDontDeadStrip on one of your TView subclasses. If you need to find out more precisely which components are missing, you can break on the Failure routine with a debugger.

Q *How many new Inside Macintosh books will there be by the time all the new technologies are documented?*

A How much shelf space do you have?

For more information on universal interfaces, see the article "Making the Leap to PowerPC" in this issue.•

Have more questions? Need more answers? Take a look at the Macintosh Q&A Technical Notes on this issue's CD and in the Dev Tech Answers library on AppleLink.•

KON & BAL'S PUZZLE PAGE

SOUNDS LIKE TROUBLE

See if you can solve this programming puzzle, presented in the form of a dialog between Konstantin Othmer (KON) and Bruce Leak (BAL) — and a special guest, Apple summer intern Mike Dodd. 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

Mike Hey, guys. I've got one for you I bet you can't get.

KON Well, I haven't been able to get you over to the poker game; maybe I can beat you here. Wanna put your summer salary on it, schoolboy?

BAL Careful, Mike. You're talking to trained professionals here.

Mike OK. We have this true multimedia application that does lots of things: plays movies and sounds and does some GWorld stuff. After a while it crashes with a corrupt heap, usually SysError 33, and sometimes with a bus error. I'm using the latest QuickTime, and I have that snazzy PowerPC QuickDraw extension that installs PowerPC native QuickDraw on 68K-class machines.

BAL Sounds like some quality software you're running there. I doubt it's QuickTime's fault.

KON Yeah, and the PowerPC stuff is pretty awesome. You wouldn't stick us with some stupid application bug, so it's probably a problem with the Sound Manager. Is this that MoveHHi Sound Manager problem? MoveHHi snags the whole stack, and when the sound interrupts come in, the stack overflows into the application space, corrupting the heap. Since the Sound Manager is at a higher interrupt level than the stack sniffer VBL, it never detects the problem. Unsolved Mysteries: Solved!

100 Mike I'm using the new Sound Manager, version 3.0.

KONSTANTIN OTHMER AND BRUCE LEAK have been awarded a subcontract to debug the Ada software for the cost-reduced space station backup project, code named BALKON-4. While BAL has found that rocket science makes him nauseous, KON has taken to weightlessness like a bug to code and is thinking of opening KONstellation, the first casino in space.•

MIKE DODD is the official QuickTime perpetual summer hire, just finishing his fourth summer with Apple. He claims that someday he'll actually graduate from the University of Tennessee and get a full-time job. Mike spends a lot of his time inside MacsBug trying to make QuickTime crash less, or at least finding cool bugs to try to stump KON and BAL with.•

BAL I guess Reekes did a good job with compatibility on the new Sound Manager. He even maintained all the bugs!

Mike Reekes swears there's not a line of code the same between the new and old Sound Managers. Besides, the new Sound Manager patches MoveHHi to not use as much stack.

KON Does it happen with the old Sound Manager?

90 Mike Yep. Same thing.

BAL Hmmm. So what are the circumstances around the crash? Do you have a reproducible case?

80 Mike It seems to happen fairly randomly. But it generally occurs when I push a button that plays a sound. Sometimes it happens the first time I push the button; other times I push the button over a hundred times before it crashes.

KON Turn on heap scrambling in the application and system heaps and run something like the MemHell extension, which forces a worst-case memory scenario. That should bring the problem out more frequently. Maybe you can get a reproducible case.

BAL Yeah, and turn on A-trap recording and heap checking so that we can narrow down the problem area.

70 Mike The application is running really slowly now, but the problem doesn't happen any more frequently. Every time you crash, you notice the last trap that the application called was SndNewChannel.

KON Wait a second. The application calls SndNewChannel every time it plays a sound? It should just call SndNewChannel once at startup for each channel it needs and then keep reusing those.

BAL What happens if you fix the application?

65 Mike The problem goes away. But you haven't found the bug yet. Although calling SndNewChannel all the time may slow you down, it isn't illegal and shouldn't cause heap corruption.

BAL Is anything else going on while the sound is being played?

60 Mike The problem seems to happen only while a movie is playing. The application calls SndNewChannel, SndPlay, and SndDisposeChannel every time it wants to make a sound, but why the crash?

BAL Does the movie have sound? What happens if you turn off the sound in the movie?

55 Mike The problem goes away.

KON What's the last trap called inside SndNewChannel before the crash?

The MemHell extension is on this issue's CD.•

50 Mike MoveHHi.

KON What if you don't play the video?

45 Mike It crashes.

BAL Change the button that's doing the SndNewChannel, SndPlay, and all that other stuff; make it so it creates a bunch of handles and calls MoveHHi on them instead.

40 Mike Now the machine crashes more frequently.

BAL So now we know that MoveHHi and playing the sound in the movie have something to do with it. Make it so that when you push the button, the movie starts playing at the beginning; then do NewHandle, MoveHHi, and DisposeHandle in a loop with a counter, and keep the loop counter at location 0 so that when you crash you can see which iteration you're on. You might have to make the size of the handles vary in case the failure depends on block size or position, since that'll help spread the allocations throughout the heap.

35 Mike It seems to happen consistently on the sixty-ninth iteration of the loop, reproducibly, if I start from launching the program.

KON Go into MacsBug and put a breakpoint in the loop when the loop counter is 69.

30 Mike You hit your breakpoint and trace over the call to MoveHHi, and it works fine. If you say go, you crash a hundred iterations later.

KON Rather than use MacsBug, change the code to break on the sixty-ninth iteration. Then what happens when you trace over MoveHHi?

25 Mike It works fine.

BAL OK. Change the code to break on the seventieth iteration.

20 Mike You get to the breakpoint at 70 and everything is fine.

KON So somehow this thing is timing sensitive. Have the program compare with a really big number and see when the heap goes bad. Then change the number so that it breaks right before the problem code.

15 Mike When you break and trace, it doesn't happen.

BAL What if I turn off interrupts during my MoveHHi loop?

10 Mike It works fine, but you only hear the first half second of sound from the movie. Since interrupts are blocked, the Sound Manager can't call back to QuickTime to get the next piece, which QuickTime has queued up in the mean time.

KON Great, so what you're saying is we can now break right before the MoveHHi that would cause it to happen if we didn't break there.

SCORING

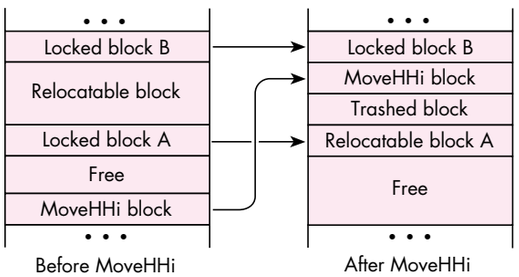
75–100 We'd award you a copy of *Debugging Macintosh Software*, but obviously you don't need it.

50–70 We'd award you a copy of *Debugging Macintosh Software*, but then KON would miss out on his royalties.

25–45 Only scores from your first reading count.

5–20 Outfoxed by a summer intern. Care to join our poker game?•

BAL Right, so go ahead and break there. Dump the state of the heap and log it to a file. Then do the same thing, this time comparing with a higher number and letting it trash itself, and dump the heap again. Compare the heaps and figure out what's going on.



5 Mike The interesting part of the heap before and after the MoveHHi call is shown in the figure. Before MoveHHi there was a locked block, labeled A in the figure, which is marked as relocatable afterward. The relocatable block just below locked block B is getting overwritten by the block we're calling MoveHHi on.

KON MoveHHi works by first saving the contents of the block that you're moving, then marking the block as free. Then it calls CompactMem on the heap, which bubbles all the free space up to any islands and all relocatable blocks down. Then it copies the block to the free block just before the island.

BAL And someone is coming in at interrupt time and unlocking the island, block A in the figure. Instead of remembering the location of the island, MoveHHi searches for it after the CompactMem call. Since that block was unlocked by an interrupt after CompactMem, a different block is found the second time. When MoveHHi backs up to the previous, presumably free, block and starts copying data, the heap gets trashed.

Mike Yeah, that interrupt unlocking the block was QuickTime, BAL. It turns out the Sound Manager does the same thing. Apparently the "system architects" at the time thought it was OK to call HUnlock on a locked block during an interrupt. *Not!* We fixed it by deferring all HUnlock calls until MoveHHi finishes. This was the cleanest fix; it keeps us from patching out huge parts of the Memory Manager. But we were stumped for quite a while.

KON Nasty.

BAL Yeah.

Thanks to Gary Davidian, Peter Hoddie, and Jim Reekes for reviewing this column.•

INDEX

For a cumulative index to all issues of *develop*, see this issue's CD.•

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Y