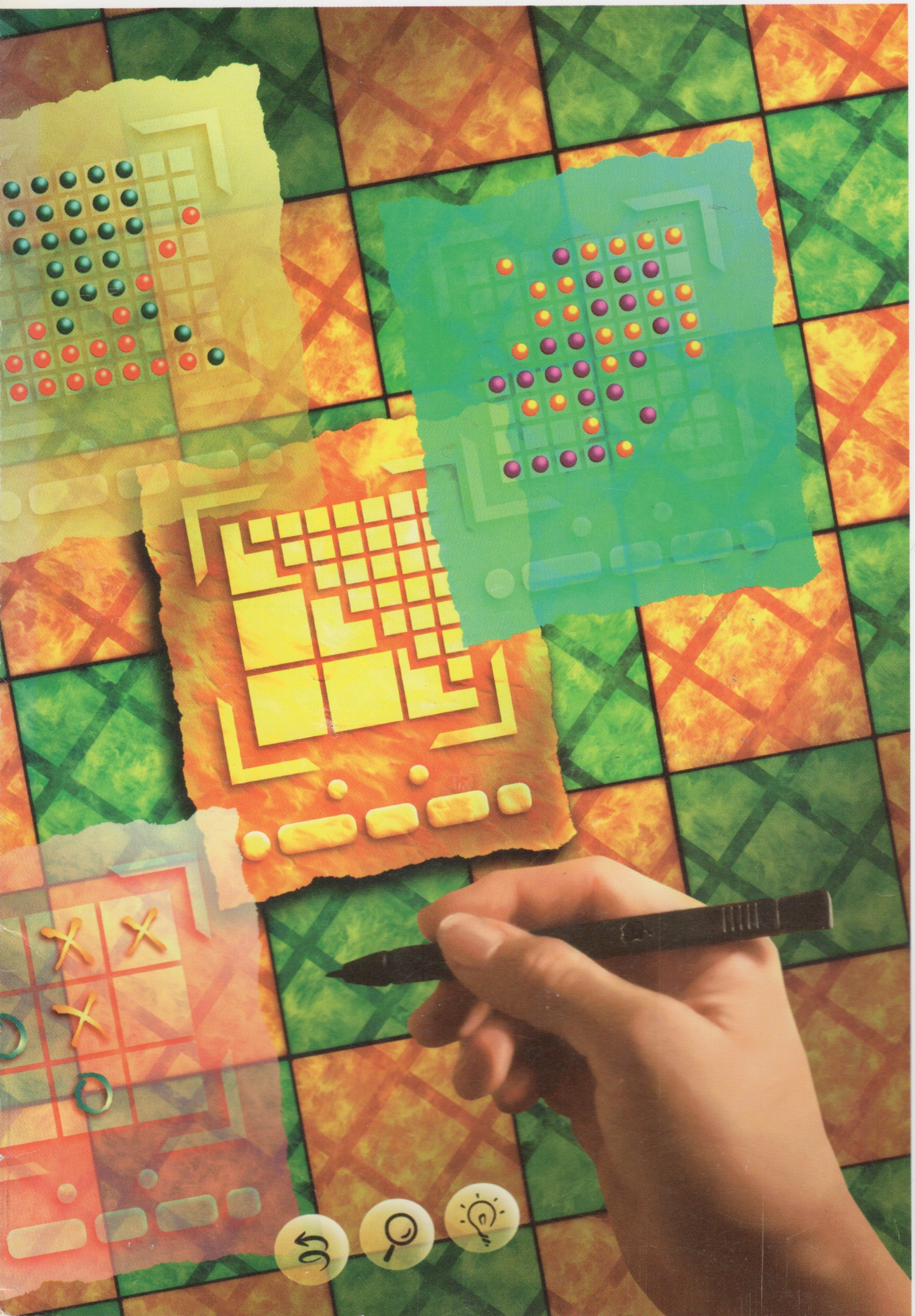
develop

The Apple Technical Journal



Issue 17 March 1994

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

This issue's CD. The *develop Bookmark* CD (or the *Developer CD Series* disc, Reference Library edition) for March 1994 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.

Macintosh Technical Notes. Where references to Macintosh Technical Notes in *develop* are followed by something like "(Memory 13)," this indicates the category and number of the Note on this issue's CD.

E-mail addresses. Most e-mail addresses mentioned in *develop* are AppleLink addresses; to convert an AppleLink address to an Internet address, append "@applelink.apple.com" to it. For example, DEVELOP on AppleLink converts to develop@applelink.apple.com on the Internet. To convert a NewtonMail address to an Internet address, append "@online.apple.com" to it.

EDITORIAL STAFF

Editor-in-Cheek *Caroline Rose*

Technical Buckstopper *Dave Johnson*

Our Boss *Greg Joswiak*

His Boss *Dennis Matthews*

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

Managing Editor *Cynthia Jasper*

Contributing Editors *Lorraine Anderson, Toni Haskell, Judy Helfand, Elaine Meyer, Rebecca Pepper*

Indexer *Marc Savage*

ART & PRODUCTION

Production/Art Director *Diane Wilcox*

Technical Illustration *Shawn Morningstar, John Ryan*

Formatting *Forbes Mill Press*

Film Services *Aptos Post, Inc.*

Prepress Production *PrePress Assembly*

Printing *Wolfer Printing Company, Inc.*

Photography *Sharon Beals*

Cover Illustration *Mark Jenkins of Rucker Huggins Design*

Online Production *Cassi Carpenter*

CAROLINE ROSE

Dear Readers,

We're excited to bring you, in this issue, *develop*'s first Newton article. Even if you're not set up for Newton development, you may find this article of interest. But it's the article on zooming windows that inspired this editorial. The subject is user interface annoyances: those cases where the application doesn't quite do what the user expects it to — as when a window zooms to an odd location — and the user has to adjust for it. So what's the big deal? Well, it all adds up. Over time, the harm to the user from compensating for these problems can be physical as well as mental. You'll end up with a customer who is suffering in more ways than one.

As Joan Stigliani puts it in her forthcoming book, *High-Tech Health: The Computer User's Survival Guide*: "Software can make you work hard and contribute to overuse if it requires a lot of mouse use — clicking and dragging, scrolling, moving the cursor back and forth across the screen — or a lot of complex key sequences or excessive keying. Software that is difficult or frustrating to use can increase stress and tension." This especially struck a chord with me, since I suffer from tendinitis caused by excessive keying and mousing. So I'm going to take advantage of this opportunity to vent my frustration (isn't that what editorials are for?). Mainly, I hope to have at least some small influence on how you design the interface for your applications in the future.

Why do so many applications lack common sense? Why, for example, shouldn't Print or Save work on my frontmost document even if the active window happens to be a dialog box? Why can't I just type Command-F followed by text to be found rather than first have to select the text that the application (for some odd reason) didn't choose to highlight in the Find dialog? Why, even on my two-page monitor, do I have to resize a teeny window for every piece of e-mail I receive, or read a mere eight lines at a time, scrolling repeatedly (excessively) to get to the end? Why, when I cut a double-clicked word and then paste it, do so few applications add spaces intelligently?

The list of user interface superfluities goes on and on; these are only the problems I encounter most frequently each day. Please, give my hands (and my mind) a break! If I may make a few suggestions:

- Don't blindly follow what other applications have done: maybe they didn't think it through well enough. There's a place for

CAROLINE ROSE (AppleLink CROSE) started writing and programming at a company called Tymshare, where she thought at first that the terminal was the computer. (She was stunned to learn it occupied a huge room in another building.) By the time computers *were* the size of terminals, Caroline was on her way to Apple to write *Inside Macintosh*. She digressed to spend five years at NeXT, where she managed the Publications group, but returned to the Apple fold to edit *develop*. Caroline owes her love of the printed word to her father, who worked for the *New York Daily News* for over 50 years. There was no greater thrill as a child than to go to the office with him and see the copy desks, darkrooms, printing presses — and, of course, the editors. She'd like to take this opportunity to say thanks, Dad, and Happy 85th Birthday!•

guidelines and precedents, but don't totally disregard common sense.

- Use the application yourself for real tasks until you're blue in the face. Be honest; what bugs you about it? Your users will be even less tolerant.
- Do extensive user testing before first ship, of course — but even after you ship, solicit feedback and incorporate it into the next release. Be sure to get feedback from experienced users, not just first-timers.
- Beware of creeping featurism: fix problems your testers or customers have with current features before adding new ones.

These suggestions are based on my own experience as the user manual writer and ad hoc product manager for the first version of the WriteNow application. John Anderson, one of the authors of WriteNow, says he thinks interface problems stem from software being too hard to write (something his next product will address) and from the related problem of software teams being too large. With many specialized programmers on a project, no one person focuses on the overall picture well enough to make the requisite intelligent decisions about the interface. Given a large team, a good product manager can make all the difference in the world. Look for someone both knowledgeable about the market and able to grasp the technical issues. As I noted about technical writing in an earlier editorial, product management isn't a job that just any smart person can do, or that CEOs or VPs should tackle in their spare time.

So I guess the moral is that more is not always better — not when it comes to features in an application or programmers on a project, and certainly not when it comes to keystrokes and mouse clicks in an interface. Please, keep it clean.

CRose

Caroline Rose
Editor

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 and the *Developer CD Series* disc.•

LETTERS

FLOATING BUGS

I liked Issue 15's article on floating windows and have successfully implemented your routines. I did, however, find one bug. Your routine HideReferencedWindow can leave the process's window list pointing to a deallocated window when there are no floating windows.

— Chester Murphy

Thanks for pointing this out. It's fixed — along with some other small bugs — in the code on this issue's CD. (It's been fixed since Issue 16's CD.)

— Dean Yu

PRAGMATIC SOLUTION

Floating windows is one of my favorite subjects. I tried to compile your WindowExtension example, using THINK C 6.0. My first problem was with USES68KINLINES. THINK C checks the #pragma parameter with the function declaration. So I tried to use A1 and D0 as parameters, but then I needed a few more #pragmas for the activate handlers. I still don't have a working version.

Also, could you explain why we need CallUniversalProc? I'm quite happy with the C syntax.

Thank you very much for doing such a good job on the article.

— Robert Puyol

The use of USES68KINLINES has changed since Issue 15 went to press. Currently, #pragma parameters aren't used any more, so you shouldn't have any problems using the header file with THINK C. Additionally, the new version of the floating windows code uses the universal header files that are discussed in the "Making the Leap to PowerPC" article in Issue 16 and are on this issue's CD.

We recommend using CallUniversalProc in your source code (or one of its specific variants, like CallActivateHandlerProc in the floating windows code) to allow for greater portability of your source code between platforms. For a detailed explanation of using CallUniversalProc and UniversalProcPtrs, check out the aforementioned article in Issue 16.

— Dean Yu

DESKTOP FILES REDUX

I liked Issue 15's Puzzle Page, about the damaged desktop files. I've seen this bug often, since I fix a lot of damaged hard disks.

You probably know this, but if the desktop files get hosed in a certain way, rebuilding them in the Finder won't fix them. You have to delete them (or rename them) to get the Finder to build new files from scratch.

AutoDoubler comes with a little utility called Desktop Reset just for deleting the desktop files.

— David Shayer

I didn't know this when I wrote that Puzzle Page, but you're right. When the desktop file gets damaged to the point where the File Manager can't open it, the Finder can't rebuild the file, so you have to throw it away or rename it.

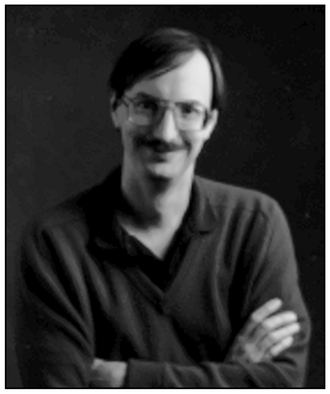
Thanks for the feedback.

— Konstantin Othmer

IF MORE OF YOU DON'T WRITE, WE'LL HAVE TO START MAKING THEM UP!
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).•

USING PROTO TEMPLATES ON THE NEWTON

Proto templates are a central feature of the Newton development environment. All Newton applications use built-in prototypes, and developers can also write their own application-specific prototypes. This article uses proto templates in the design of an application that plays a few simple games. For non-Newton developers it reveals some of the flavor of designing and writing Newton applications.



HARRY R. CHESLEY

An application often includes multiple instances of a design element, with only minor variations among them. In object-oriented systems, it's possible to share the common portions of the design among several different pieces of the application by using inheritance. On the Newton, you can do this with *proto templates*, which let you reuse the definition of a particular type of view very efficiently. Since views are the basic visual and functional elements of the Newton user interface, proto templates give you a very powerful ability to share and reuse user interface features.

Built-in proto templates supply most of the common views seen in Newton applications — push buttons, pop-up menus, checkboxes, radio buttons, and so on. In addition, developers of Newton applications can define their own proto templates. These templates allow for very compact designs, shorter development times, easier maintenance, and smaller finished applications.

In this article we develop a game application for the Newton called TapBoard. TapBoard is actually three games, each with its own style of board, rules of play, and algorithm for computer-generated moves. The games have many elements in common, and these can be abstracted into a proto template containing the shared aspects of the design. You'll find much of the code for TapBoard in the article; the complete source code can be found on this issue's CD.

If you've never written a Newton application, see "(Slightly) Inside Newton Programming" for an introduction to the development process and some of the Newton terminology that's used in this article.

HARRY R. CHESLEY (AppleLink CHESLEY1, NewtonMail CHESLEY) has spent most of the last three years in subspace (see *develop* Issue 7, "The Subspace Manager in System 7.0") working on PowerTalk templates. His return to Earth-normal space was accompanied by an interstitial stutter, resulting in a temporary doubling of his personality matrix. This allowed him to simultaneously finish PowerTalk and design and write the NewtonScript communications interface (protoEndpoints). He's better now, but still occasionally repeats himself in social situations. If he should happen to tell you a story you've heard from him before, just humor him and pretend it's all new stuff. Too much self-referential introspection could cause a substitial relapse.•

(SLIGHTLY) INSIDE NEWTON PROGRAMMING

BY GREGG WILLIAMS

Programming the Newton is *different* — and I mean that in a good sense. Once you digest the Newton documentation, you can create a simple application and have it running on a Newton in about 15 minutes. The Newton development process encourages lots of quick code/compile/debug cycles — good for those of us who need positive reinforcement on a regular basis — and the cycle is short enough that you don't notice a delay.

OVERVIEW

Before I describe how a Newton program works, here's an overview of the development process. To create a Newton application, all you need is the Newton Toolkit (available from APDA), a serial cable, a 68030 Macintosh with 8 MB of memory and 32-bit addressing, and a Newton. You create your program with the Toolkit, compile it, and then download it to a Newton that's connected to your Macintosh through a serial port. Your completed application appears in the Extras drawer, just like any other third-party Newton application.

Using tools from a floating window called the *layout palette*, you draw the layout of your application — what its screens should look like — in *layout windows*. Not only can you reuse the dozens of user interface definitions that the Newton Toolkit supplies, you can create your own *custom proto templates* (more on proto templates below). Each layout or custom proto template is stored as a separate file; all of these files, plus optional files that contain the Macintosh-style resources your program needs, are grouped into what's called a *project*.

Once you've drawn all your views, you can open a *browser window* (familiar to users of object-oriented languages like Smalltalk and LISP). In the browser window, you can add and modify both code and data associated with the objects in your program. The code you write is in a new language called *NewtonScript*, which is a simple but sophisticated symbolic language with a Pascal-like syntax.

Most of the (usually short) routines you'll write execute when triggered by a user action, like the user's tapping a button or writing in a designated area, or by a system action. You're already acquainted with this event-driven approach from your experience in programming for the Macintosh. And if you've been doing object-oriented programming, the idea of a network of cooperating software objects, rather than a hierarchy of routines executed by a processor that's always in control, is also familiar to you.

FRAMES AND SLOTS

One idea you may not be familiar with, depending on whether you've used object-oriented languages, is that of *frames*. Artificial intelligence fanatics will probably kill me, but I think of a frame as just a record of data, and *slots* as what most of us call record fields. However, slots in NewtonScript are more versatile than record fields in several ways. First, slots aren't limited to one type of data. (Like Newton variables, slots aren't typed and can hold, for example, an integer one minute and a frame the next.) Second, you can arbitrarily add or remove slots from a frame at any time, including during program execution. Third, you can access slots indirectly through *path expressions*; these allow you to store part of a slot's pathname in a variable, thus letting the contents of a variable determine which slot gets accessed. In NewtonScript, a frame looks like this:

```
{ slotname1: value1, slotname2: value2, ...,
   slotnameN: valueN }
```

Slot names can be omitted or mentioned in any order. By using the slot names _proto and _parent, you can create data structures that exhibit Newton's flavor of object-oriented behavior, as discussed later in this article.

To access the data in a frame's slot, you use the notation

```
framename.slotname
```

TEMPLATES, VIEWS, AND PROTOS

Everything you see on the Newton screen is composed of *views*. A view can display, among other things, a picture, a paragraph of text, an area for writing or drawing, an on-screen keyboard, a calendar-month page, a pop-up list, or a gauge (which is like a horizontal thermometer).

Many of the things that you see on the screen are standard prefabricated user interface elements built into the Newton ROM and available to every Newton developer. These are called *proto templates*, or *protos* for short, and they include six different kinds of buttons and checkboxes, protoSliders (like a linear slider light switch), protoRolls (which allow information to scroll vertically off the screen), and different kinds of labels, borders, and standard interface elements. Each tool on the Newton Toolkit layout palette allows you to lay out a view or a proto.

Views can contain views inside them, and in fact, a proto is a predefined hierarchical grouping of views that behaves in a certain way. You can create your own custom protos by drawing them in layout windows.

In NewtonScript, you can describe a view as a frame called a *template*; the Newton later creates a view from the template at run time. The slots in a template represent the view's data and the functions that implement the behavior you have to add. (Most views and protos have behavior built in, and the built-in behavior of a proto is often quite extensive.) When your application executes, a template (which cannot be changed) is used to create the data structure in RAM that corresponds to the visual representation of the view — that is, what you see on the Newton's screen.

To send a message to a view (that is, to invoke one of its methods), the syntax is

```
view:messagename(arg1, arg2, ..., argN)
```

(The function involved is the *method*, while the name used to invoke it is the *message*. You send a message to an object, and that causes the associated method to execute.)

SYSTEM MESSAGES

As with the Macintosh, Newton applications are driven by user events — the user taps or draws on the Newton screen in different places — as well as by system events. The Newton sends your program *system messages*, which trigger their corresponding methods. In some cases, you let the built-in Newton methods do their work; in many others, you write your own. There are 25 or so system messages you need to know about. Here are a few of them, and when the corresponding methods execute:

- viewChangedScript: executes when a view slot is changed using the SetValue function or when certain functions change a view directly
- viewDrawScript: executes when a view needs to be drawn
- viewStrokeScript: executes when a user writes inside a view
- viewSetupFormScript, viewSetupChildrenScript, viewSetupDoneScript: execute at specific points during the setup of a view, before the view is displayed
- viewIdleScript: executes periodically
- viewQuitScript: executes just before a view is disposed of

Some system messages pertain to particular views — for example, buttonClickScript (for buttons), keyPressScript (for keyboard views), and monthChangedScript (for the monthly calendar view).

AND THAT'S JUST THE BEGINNING. . .

There's a lot more to programming the Newton. You can find out more by reading the manuals that come with the Newton Toolkit. Be sure to read any errata sheets, release notes, and "read me" files; I overlooked one and missed a piece of information that would have saved me several days' work!

For more information about views, see the "Views" chapter of the *Newton Programmer's Guide.*•

GAMES WE'RE GOING TO PLAY

TapBoard plays three games: Tic-tac-toe, Gomoku, and Reversi. The user chooses one of the games from a set of radio buttons, a game board is displayed, and the user moves by tapping a square on the board. The application responds with a countermove. This process repeats until one player wins or the game ends in a tie. When the user closes TapBoard, the application remembers the state of the board — which game is being played, where the pieces are placed, whose turn it is, and the game's outcome. When TapBoard is reopened, it restores the state so that the user can continue the game where it left off.

So, common elements shared by the games include the following:

- displaying the board to the user
- adding a new move to the board display
- acknowledging the user's tap with an appropriate sound
- tracking the stroke to decide whether it ended on the same square as it started on
- figuring out which square it was, and whether the move is valid
- recording the move
- checking it to see if it's a winning or tying move
- uncovering a good move when it's TapBoard's turn
- saving the state of the board when the user closes the game
- restoring the state when the game is reopened

All these common actions can be abstracted into a prototype that each specific game can then inherit from (as described later in the section "The protoBoard Proto Template"). But since the games are different, each also has its own rules and possibly its own board, and each requires its own algorithm for finding good moves for the computer; details follow.

TIC-TAC-TOE

You undoubtedly know this game, but we'll describe it briefly: Tic-tac-toe is played on a 3 x 3 board with players taking turns making X's and O's. The first player to get three in a row horizontally, vertically, or diagonally wins. If neither player gets three in a row, the game is tied. Figure 1 shows a typical game of Tic-tac-toe. X gets three on the diagonal and wins.

Tic-tac-toe is a simple game to master. There are a few easy heuristics, such as always take the center square if you can, and with a little thought it's not hard to see several moves ahead.

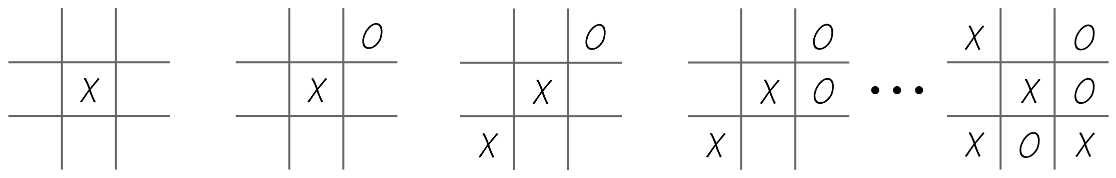


Figure 1
Tic-tac-toe

TapBoard's algorithm for determining a good Tic-tac-toe move involves doing a two-move look-ahead, combined with some simple heuristics. The look-ahead tries all possible moves, then tries all possible answering moves. The heuristic gives greater weight to taking the center and corner squares. This algorithm doesn't play a perfect game of Tic-tac-toe, but this is actually an advantage, for two reasons: it pulls players into the game by giving the impression that the computer is an easy target, and it provides a game that's playable by younger users (my five-year-old daughter loves it).

GOMOKU

Gomoku is played on an 8 x 8 board. Players take turns placing pieces on the board, and the first player to get five in a row horizontally, vertically, or diagonally wins. Figure 2 shows a short game of Gomoku won by white (a real game would of course be played more defensively than this one, which is for demonstration purposes only).

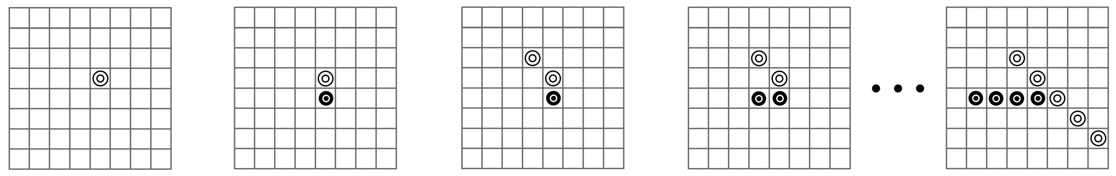


Figure 2
Gomoku

Gomoku is more challenging than Tic-tac-toe. The larger game board and longer winning sequence make for many more combinations, and the game virtually requires that the winner "sneak up" on the loser, rather than just going for a simple sequence of five pieces, which the opponent can easily detect and prevent.

TapBoard's approach to finding a good Gomoku move involves making three passes over the board: the first looks for winning moves; the second looks for situations where the user has three or more in a row and tries to block those; the third looks for situations where TapBoard has three or more in a row and tries to add to those. This algorithm plays a defensive game that's hard to beat.

REVERSI

Reversi is also played on an 8 x 8 board. Four pieces are placed in the center of the board as shown in Figure 3A, and the players take turns placing pieces, trying to trap the opponent's pieces between the new piece and an existing one horizontally, vertically, or diagonally. Figure 3B shows one of the squares that would be a legal first move for white. The pieces that are "trapped" change color, or reverse — hence the name of the game. Figure 3C shows the result of the move in 3B.

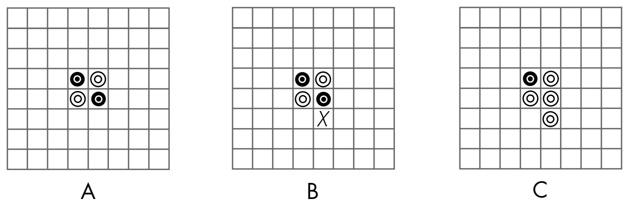


Figure 3
Reversi

Play continues until no legal moves are left. The player with the most pieces wins. If both players have the same number of pieces, it's a tie. Because of the reversing pieces, the situation can change suddenly and dramatically. Often one player looks like the clear winner until near the end of the game, when the other player suddenly surges ahead and wins.

TapBoard looks for a good move in Reversi by checking every possible move, counting the number of user pieces that will be converted, and then modifying the counts based on heuristics concerning plays along the edges of the board. This algorithm plays the game quite well, and can often come on surprisingly strong at the end of the game.

THE TAPBOARD APPLICATION

The layout of the TapBoard application is shown in Figure 4. The center of the screen is the game board, and there are three different game board templates corresponding to the three games. Only one is displayed at a time. Below the board are radio buttons indicating whose turn it is. Underneath these are pictures of the pieces for the user and for the application, which change from game to game. Below them is a set of radio buttons with the names of the games, from which the user chooses. At the bottom of the screen is a set of buttons including a time and battery status button, buttons for New Game, Help, and Credits, and a close box.

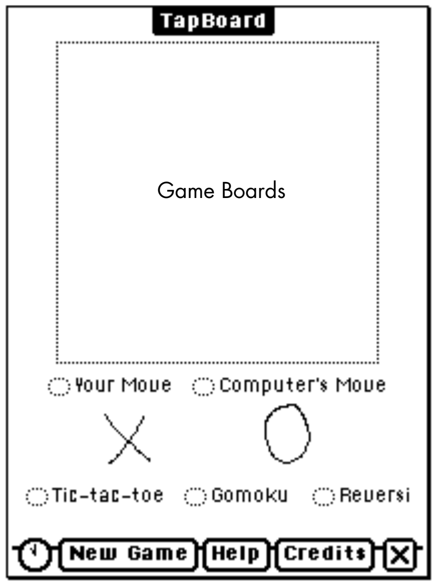


Figure 4
TapBoard Application Screen Layout

Besides the games themselves, the application has to take care of the following:

- selecting which game to play
- showing whose turn it is, and letting the user choose who goes first
- displaying help and credits
- triggering save and restore operations when the application closes and reopens
- removing the saved state from memory when the application is removed from the Newton

Most of the TapBoard application is in the application template and the game boards. The buttons are in the application template. The core functionality of the application, playing the games, is in the game boards and in protoBoard, the proto template from which all three games inherit. (Newton inheritance is a little different from what you might be used to; for more information, see “Proto and Parent Inheritance.”)

THE PROTOBOARD PROTO TEMPLATE

The protoBoard proto template takes care of several functional areas. In some cases, it does everything needed by the game boards that derive from it. In other cases, one or more of the game boards must override part of the proto template functionality —

PROTO AND PARENT INHERITANCE

When you draw a view inside another view, the one inside is said to be a *subview* or a *child view*. When you draw nested views in a layout window in the Newton Toolkit, you're doing the equivalent of writing code that is a frame, with each piece of data or method having its own slot.

Unless it's at the end of its chain, every view has a _proto slot, which points to the proto from which it inherits its behavior. Similarly, every view (except the "top" one) has a _parent slot, which points to the view that contains it (that is, its *parent view*).

Here's where things get interesting. A view exists in RAM while your program is executing; it can have its own slots, which are also in RAM. If some code tries to access a slot that the view doesn't have, the view looks for that slot in the view's proto. If its proto doesn't have that slot, the proto looks for the slot in the proto's proto (if it has one), and so on up the line of proto "ancestors." The value that's ultimately returned is used as if it were actually a slot in RAM belonging to that view.

But wait — there's more! You also have parent inheritance: a view can get a slot value from one of its parent views. If the desired slot isn't found in any proto, the search continues (from above) with the view's parent and, if necessary, its protos. (The process is a bit different when changing slot values; the only slot values you can change are in the view or one of its ancestors — all of these slots, of course, are in RAM. For details, see the "Working With Proto Templates" chapter of the *NewtonScript Programming Language* manual.)

Why two kinds of inheritance? Two answers: ROM and application size. The reason for views and protos in the first place is to minimize the amount of code in your application and so make it as compact as possible. Because views and protos are in ROM, though, you can't change their slots. You need another mechanism to override their default values, and that's where proto inheritance comes in.

Proto inheritance shrinks application size by minimizing the amount of code associated with individual instances of often-used "building blocks." Parent inheritance shrinks application size by allowing related elements in your program to share common data or behavior. (For example, three radio buttons in a cluster may share a button-click method that's different from that of other buttons on the same screen.)

In addition to sending a message to a certain view, you can send a message that starts with the current view and checks protos, parents, and parent protos until it finds a view or proto template that has a slot with the same name as the message. The associated method, wherever it comes from, is then applied to the current view. The syntax is

```
:messagename(arg1, arg2, ..., argN)
```

This is a very brief overview of a software architecture that has many implications. To fill in the gaps, read the Newton documentation.

— GW

and several slots are *designed* to be overridden. In still other cases, a protoBoard function uses data slots defined in the game boards as input.

BOARD STATE AND SHAPE

The protoBoard proto template maintains a two-dimensional array, named boardArray, that remembers where pieces have been placed on the board. The array is actually one entry larger than the board in each direction. This oversizing can be handy in the algorithms used to find a good move for the application.

Each space in the array can have one of four values: empty, Newton piece, user piece, or edge of the board. To simplify the algorithms used to compute the application's moves, we use 1 for the user and -1 for the Newton, so that if *p* is a piece of one type, -*p* is the opposing type of piece. Empty spaces are filled with nil, and board edges are 0 — neither nil nor a valid Newton or user piece value. These same values are used to keep track of whose turn it is and who has won (if anyone). We define constants for each of these potential values:

```
constant kEmptySquare := nil;
constant kNewtonPiece := -1;
constant kUserPiece := 1;
constant kBoardEdge := 0;
constant kTieWinner := 0;
```

Here's the portion of protoBoard's viewSetupFormScript that creates boardArray:

```
protoBoard.viewSetupFormScript := func()
begin
   . . .
   // Make the board array; we make it one entry larger in each direction
   // than the board, which is nice sometimes when figuring out moves.
   boardArray := Array(squaresWide+2, kEmptySquare);
   local i;
   for i := 0 to squaresWide+1 do
      begin
         boardArray[i] := Array(squaresHigh+2,
                                if (i = 0) or (i = squaresWide+1) then
                                   kBoardEdge else kEmptySquare);
         boardArray[i][0] := kBoardEdge;
         boardArray[i][squaresHigh+1] := kBoardEdge;
      end;
   // Reset the number of squares left.
   squaresLeft := squaresWide * squaresHigh;
   // No winner yet.
   winner := nil;
   // Do any game-specific setup.
   :setupBoard();
   . . .
end
```

We initialize squaresLeft to the number of squares on the board so that TapBoard can determine whether the board is full without having to check every square. We also set winner to nil. When the game is finished, the winner slot is set to kUserPiece (user won), kNewtonPiece (Newton won), or kTieWinner (tie). This allows for a very quick check on whether the game is over and who won.

The setupBoard function is called to do any game-specific board setup. For example, Reversi needs to place four initial pieces on the board. The default setupBoard function defined in protoBoard does nothing; inheritors of protoBoard override it as needed.

Two slots, squaresWide and squaresHigh, must be defined by any protoBoard inheritor to determine the width and height of the board. They're used throughout protoBoard — as in viewSetupFormScript — and in the following utility functions. (Note that LocalBox is a function that returns a rectangle having the width and height of the view in its right and bottom slots, respectively.)

```
// The height of a square:
protoBoard.squareHeight := func()
begin
   return :LocalBox().bottom div squaresHigh;
end

// The width of a square:
protoBoard.squareWidth := func()
begin
   return :LocalBox().right div squaresWide;
end

// The bounds of a square:
protoBoard.squareBounds := func(x, y)
begin
   local width := :squareWidth();
   local height := :squareHeight();
   // RelBounds takes a top and left coordinate, a width, and a height
   // and returns a rectangle.
   return RelBounds((x-1)*width+1, (y-1)*height+1, width-1, height-1);
end

// Which square (1..squaresWide) contains coordinate x (or zero if none):
protoBoard.squareOfX := func(x)
begin
   local gb := :GlobalBox();
   if (x < gb.left) or (x > gb.right) then return 0;
   else return ((x - gb.left) div :squareWidth()) + 1;
end
```

The utility functions are usable only at viewSetupChildrenScript time and later because they use the LocalBox function. You couldn't use them, for instance, in viewSetupFormScript.•

```
// Which square (1..squaresHigh) contains coordinate y (or zero if none):
protoBoard.squareOfY := func(y)
begin
   local gb := :GlobalBox();
   if (y < gb.top) or (y > gb.bottom) then return 0;
   else return ((y - gb.top) div :squareHeight()) + 1;
end
```

DRAWING THE BOARD

The drawing of the board itself is created once and stored in the slot backgroundDrawing. This drawing is then displayed by viewDrawScript.

```
protoBoard.viewDrawScript := func()
begin
   :DrawShape(backgroundDrawing, nil);
end
```

The backgroundDrawing slot is built in viewSetupDoneScript, which is called just before the view is shown on the screen. The default function supplied in protoBoard draws a closed set of squares for the board. This is appropriate for Gomoku and Reversi but is overridden by Tic-tac-toe, which needs an open grid.

```
protoBoard.viewSetupDoneScript := func()
begin
   // Build the board display. This builds a closed set of squares,
   // but can be overridden.
   local height := :LocalBox().bottom - 1;
   local width := :LocalBox().right - 1;
   backgroundDrawing := [];   // Empty array
   for x := 0 to width by :squareWidth() do
      AddArraySlot(backgroundDrawing, MakeLine(x, 0, x, height));
   for y := 0 to height by :squareHeight() do
      AddArraySlot(backgroundDrawing, MakeLine(0, y, width, y));
end
```

ADDING PIECES

Besides an entry in boardArray, each piece has a subview within the board of class clPictureView, which displays the piece on the board. Adding a piece to boardArray, adding the corresponding subview, and adjusting squaresLeft are the responsibility of the addPiece function.

```
protoBoard.addPiece := func(p, x, y)
begin
   // Mark the new piece in boardArray.
   boardArray[x][y] := p;
   // Check if there's already a view there in the view list.
   local bounds := :squareBounds(x, y);
   local i := if p = kUserPiece then player1Piece else player2Piece;
   local v;
   // ChildViewFrames returns an array containing all child views.
   foreach v in :ChildViewFrames() do
      if (v.viewBounds.top = bounds.top) and
            (v.viewBounds.left = bounds.left) then
         begin
            // If there is, replace the icon and redisplay.
            SetValue(v, 'icon, i);
            return;
         end;
   // One less square available.
   squaresLeft := squaresLeft - 1;
   // Create, add in, and display the new view.
   AddStepView(self,
               {viewClass: clPictureView, viewBounds: :squareBounds(x, y),
               viewFlags: vVisible, icon: i}):Dirty();
end
```

To determine the picture to display, addPiece looks in slots player1Piece and player2Piece, which contain pictures for the user and for the computer. The default versions supplied in protoBoard are suitable for Reversi and Gomoku. Tic-tac-toe overrides these slots to provide an X and an O.

Note that this function checks to see if there's already a piece on that square. If there is, it simply changes the picture for the piece rather than creating a new view. In most games, moving on a square that already has a piece on it is illegal. But in Reversi, we often replace existing pieces with pieces of the opposite color.

MAKING MOVES

Checking the legality of a move, deciding whether it's a winning or losing move, and then switching the state of whose turn it is, are done in the move function. This is the function to call to actually make a move.

```
protoBoard.move := func(p, x, y)
begin
   // Check if this is a reasonable thing to do.
   if :isTurn(p) and :validMove(p, x, y) then
      begin
         // Add the piece to the board.
         :addPiece(p, x, y);
         // If this was a winner, let the user know.
         if :winningMove(p, x, y) then
            begin
               winner := p;
               :announceWin(p);
            end
         // If this was a tie-maker, let the user know.
         else if :tieGame() then
            begin
               winner := kTieWinner;
               :turn(kTieWinner);
               :announceWin(kTieWinner);
            end
         // Switch whose turn it is.
         else :turn(-p);
      end;
end
```

The functions isTurn, announceWin, and turn are global application functions defined in the application template; we'll get to them later. The functions validMove, winningMove, and tieGame are game-specific and are defined in the protoBoard inheritors. We provide default versions:

```
protoBoard.validMove := func(p, x, y)
begin
   // If it's an empty space, it's legal to move there.
   // This function may be overridden.
   return boardArray[x][y] = kEmptySquare;
end

protoBoard.winningMove := func(p, x, y)
begin
   // By default, the computer never wins and the user wins when the
   // board is full. This is always overridden, but we leave it in
   // because it can be handy during the early stages of developing
   // a new game.
   if p = kNewtonPiece then return nil
   else return squaresLeft = 0;
end

protoBoard.tieGame := func()
begin
   // It's a tie if there's nothing left to do. This can be overridden.
   return squaresLeft = 0;
end
```

THE USER'S MOVES
The user moves by tapping on the board. When the tap first occurs, viewClickScript is called. This function turns off ink and plays a sound to let the user know clearly that the tap was hard enough. The actual move is recorded in viewStrokeScript, which is called after the user lifts the pen from the screen.

```
protoBoard.viewClickScript := func(unit)
begin
   // No ink (we're tapping, not drawing).
   InkOff(unit);
   // Make a nice little click to give the user warm fuzzies.
   PlaySound(ROM_click);
   // But let the normal processing handle tracking and such.
   return nil;
end

protoBoard.viewStrokeScript := func(unit)
begin
   // Find out where we clicked to start with.
   local originalX := :squareOfX(GetPoint(firstX, unit));
   local originalY := :squareOfY(GetPoint(firstY, unit));
   // If we ended where we started, make the move.
   if (originalX <> 0) and (originalY <> 0) and
         (originalX = :squareOfX(GetPoint(finalX, unit))) and
         (originalY = :squareOfY(GetPoint(finalY, unit))) then
      :move(kUserPiece, originalX, originalY);
   return true;
end
```

We return nil from viewClickScript to say we didn't handle it, so the system processes the stroke for us. But we return true from viewStrokeScript because here we did handle the stroke and don't want the system to do anything more.

THE COMPUTER'S MOVES
It's the computer's turn to move after the user has moved, or because the user tapped the Computer's Move radio button. Rather than put code in each of these places, we simply set up an idle method called viewIdleScript in protoBoard that checks to see if it's the computer's turn and then makes its move.

This is not the most efficient approach because it involves checking periodically while the user is thinking, but it's quite simple (and shows how to set up idle methods). Since it only checks every quarter of a second, it doesn't actually use much CPU or battery power.

```
protoBoard.viewSetupFormScript := func()
begin
   . . .
   // Have our idle method called.
   :SetUpIdle(250);
end

protoBoard.viewIdleScript := func()
begin
   // If we are visible, and it's the computer's turn, and there's no
   // winner...
   if Visible(self) and :isTurn(kNewtonPiece) and (winner = nil) then
      begin
         // Put up the "Working..." display, and figure the computer's
         // move.
         :startWorking();
         :makeComputerMove();
         :stopWorking();
      end;
   // Try again in a quarter of a second.
   return 250;
end

protoBoard.makeComputerMove := func()
begin
   // By default, we just do something random.
   // This is always overridden.
   :makeRandomMove(kNewtonPiece);
end

protoBoard.makeRandomMove := func(p)
begin
   // Try ten times to find a reasonable random move.
   local i, x, y;
   for i := 1 to 10 do
      begin
         x := Random(1, squaresWide);
         y := Random(1, squaresHigh);
         if :validMove(p, x, y) then
            begin
               :move(p, x, y);
               return;
            end;
      end;
   // If that doesn't work, just pick the first linear move.
   for x := 1 to squaresWide do
      for y := 1 to squaresHigh do
         if :validMove(p, x, y) then
            begin
               :move(p, x, y);
               return;
            end;
end
```

SAVING AND RESTORING STATE

Between invocations of TapBoard the application needs to save its state, so that it can restore the state when the user reopens the application. Permanent data on the Newton is stored in one or more database-like objects called *soups* (for more on soups and related concepts, see "Soups"). Since we only need to save a fairly simple set of state information, we can put it into the system configuration and preferences soup. This soup is named "System" and contains information such as the user's name and Newton configuration options.

SOUPS

If frames are sort of like records, then soups are like what we normally think of as files — both hold collections of data. But soups — like frames — do so in a way that's more relaxed and free form. To use a food analogy, if traditional files are like a stack of sugar cubes, then soups are like, well, soup.

Soups store data in a general format that doesn't limit their usefulness to the application that created them. For example, any Newton application can access all the names in the Newton's built-in address book because the names are in the same soup.

Soups engender their own terminology. Here are a few terms you need to know:

- Entry: An *entry* points to a "record" of data. When an entry is accessed as a frame, the actual data is constructed in RAM and stays there until it's no longer referenced.
- Store: A *store* is a place where data physically resides. The Newton has a single built-in store, which is in RAM; an installed PCMCIA RAM or ROM card would be another store.
- Query: To retrieve an entry from a soup, you perform a *query* on it. This returns a cursor that represents the criteria specified by the query.
- Cursor: A *cursor* (also called a *cursor object*) points to the first entry that matches the query. To read other matching entries, you send the cursor Next and Prev (previous) messages, which "move" the cursor to point to other matching entries. Cursors are dynamic, that is, they find the next entry in the soup as it exists when the cursor is moved.

Soups are one of the uniquely new things about the Newton, and are a topic unto themselves. To learn more, see the documentation that comes with the Newton Toolkit.

— GW

The process of saving and restoring the state is triggered in the application view, as we'll discuss later. But the bulk of the actual work is done in protoBoard.

```
protoBoard.saveState := func()
begin
   // Get the existing state entry, if any.
   local stateEntry := :getStateEntry();
   // If there isn't one yet, make one.
   // Note: GetStores()[0] returns the built-in store;
   // GetSoup(ROM_SystemSoupName) returns the "System" soup.
   if stateEntry = nil then
      stateEntry := GetStores()[0]:GetSoup(ROM_SystemSoupName)
                                   :Add({Tag: kPackageName});
   // If we can't make one, well, uh, let's just forget the whole thing.
   if stateEntry = nil then return;
   // Build an array of pieces and their positions from boardArray.
   local x, y;
   local pieces := [];
   local ba := boardArray;
   for x := 1 to squaresWide do
      for y := 1 to squaresHigh do
         if ba[x][y] <> kEmptySquare then
               AddArraySlot(pieces, {player: ba[x][y], x: x, y: y});
   // Remember which game this is, the piece positions, whose turn it is,
   // and the winner.
   stateEntry.name := name;
   stateEntry.pieces := pieces;
   stateEntry.whichTurn := :whoseTurn();
   stateEntry.winner := winner;
   // Tell the soup to save the changed entry.
   EntryChange(stateEntry);
end

protoBoard.restoreState := func(stateEntry)
begin
   // For each piece stored in the state entry, add it to the board.
   local p;
   foreach p in stateEntry.pieces do
      :addPiece(p.player, p.x, p.y);
   // Set whose turn it is.
   :turn(stateEntry.whichTurn);
   // Set the winner, if there is one.
   winner := stateEntry.winner;
end
```

The constant kPackageName is the name of the application concatenated with a registered signature. This string is unique, ensuring that we don't try to use an entry in the soup that is already used by another application:

```
// Who we are:
constant kAppSymbol := '|TapBoard:Chesley|;
constant kPackageName := "TapBoard:Chesley";
```

The function getStateEntry is defined in the application template:

```
TapBoard.getStateEntry := func()
begin
   // Find our one-and-only entry in the System soup, if there is one.
   return Query(GetStores()[0]:GetSoup(ROM_SystemSoupName),
                {type: 'index, indexPath: 'tag, startKey: kPackageName,
                 validTest: func(item) StrEqual(item.tag,
                 kPackageName)}):Entry();
end
```

LET THE GAMES BEGIN

Each game inherits from protoBoard all of the functionality described in the previous section. Now we only need to define the details of the game — the size of the playing board, what the pieces look like, what the valid moves are, and the algorithm for figuring out the computer's moves.

TIC-TAC-TOE

For Tic-tac-toe, we provide the name of the game and the board size and override the default piece pictures as follows:

```
tictactoe := { ... name: "Tic-tac-toe", squaresWide: 3, squaresHigh: 3,
                    player1Piece: TapBoard.rsrc:XPicture,
                    player2Piece: TapBoard.rsrc:OPicture ... }
```

We override the board drawing in viewSetupDoneScript.

```
tictactoe.viewSetupDoneScript := func()
begin
   // Make an open cross-hatch (the default function does a closed board).
   local height := :LocalBox().bottom - 1;
   local width := :LocalBox().right - 1;
   local xIncr := :squareWidth();
   local yIncr := :squareHeight();
   backgroundDrawing := [];
   for x := xIncr to width - xIncr by xIncr do
      AddArraySlot(backgroundDrawing, MakeLine(x, 0, x, height));
   for y := yIncr to height - yIncr by yIncr do
      AddArraySlot(backgroundDrawing, MakeLine(0, y, width, y));
end
```

We can use the default tieGame function, which says the game is a tie if all the squares are used, but we need to override winningMove.

```
tictactoe.winningMove := func(p, x, y)
begin
   local ba := boardArray;
   return
      ((ba[x][1] = p) and (ba[x][2] = p) and (ba[x][3] = p)) or
      ((ba[1][y] = p) and (ba[2][y] = p) and (ba[3][y] = p)) or
      ((ba[1][1] = p) and (ba[2][2] = p) and (ba[3][3] = p)) or
      ((ba[3][1] = p) and (ba[2][2] = p) and (ba[1][3] = p));
end
```

We also need to override makeComputerMove:

```
tictactoe.makeComputerMove := func()
begin
   local moves := [];
   local bestScore := -1000;
   local newScore;
   local x, y;
   // Try each board position.
   for x := 1 to squaresWide do
      for y := 1 to squaresHigh do
         if boardArray[x][y] = kEmptySquare then
            begin
               // Look ahead to score this move.
               newScore := :tryMove(kUserPiece, kNewtonPiece, x, y);
               // If this is the best one yet, remember only it.
               if newScore > bestScore then
                  begin
                     moves := [];
                     bestScore := newScore;
                  end;
               // If it's tied for best move, remember it too.
               if newScore = bestScore then
                  AddArraySlot(moves, {mvx: x, mvy: y});
            end;
   // If there are any good moves...
   if Length(moves) > 0 then
      begin
         // Make the move.
         local move := moves[Random(0, Length(moves)-1)];
         :move(kNewtonPiece, move.mvx, move.mvy);
      end
   // If there are no good moves, make a random one and pray.
   else :makeRandomMove(kNewtonPiece);
end

tictactoe.tryMove := func(d, p, x, y)
begin
   // First, guess based on heuristics.
   // Note: We use a quoted array here to save execution time; quoting
   // it means there will be only one copy -- without the quote a new
   // one would be constructed each time at run time. Of course, this
   // also means we can't change the contents, but we don't want to.
   local score := '[5, 0, 5,   5, 10, 5,   5, 0, 5][x+x+x+y-4];
   // Make the move internally (we'll retract it later).
   local ba := boardArray;
   ba[x][y] := p;
   squaresLeft := squaresLeft - 1;
   // If it's a winner, great, give it a high score.
   if :winningMove(p, x, y) then score := 100;
   // If there's anything to look ahead to, do it.
   else if (squaresLeft <> 0) and (d > 0) then
      begin
         local worstResponse := 1000;
         local newResponse;
         local x2, y2;
         // Try every board position.
         for x2 := 1 to squaresWide do
            for y2 := 1 to squaresHigh do
               if ba[x2][y2] = kEmptySquare then
                  begin
                     // How good is this one?
                     newResponse := :tryMove(d-1, -p, x2, y2);
                     // If it's a loser, give up quick.
                     if newResponse >= 100 then
                        begin
                           ba[x][y] := nil;
                           squaresLeft := squaresLeft + 1;
                           return -100;
                        end;
                  // If it's the least bad one so far, remember that.
                  if newResponse < worstResponse then
                     worstResponse := newResponse;
               end;
         score := score - worstResponse;
      end;
   // Retract the move.
   ba[x][y] := kEmptySquare;
   squaresLeft := squaresLeft + 1;
   return score;
end
```

That's it. As you can see, most of the real work was done by protoBoard.

GOMOKU

For Gomoku, we provide the name and board size but leave the default piece pictures and board drawing.

```
gomoku := { ... name: "Gomoku", squaresWide: 8, squaresHigh: 8 ... }
```

The tieGame function in protoBoard is fine, but winningMove needs to be overridden. We also need to override makeComputerMove and makeRandomMove (which makeComputerMove calls) because the default version makes some really stupid moves in the case of Gomoku (you shouldn't move on the edge of the board if you can avoid it). You can find the code for these functions, along with the rest of the source code, on this issue's CD. Again, most of the work was done by protoBoard.

REVERSI

For Reversi, as for Gomoku, we define the name and board size but use the default piece pictures and board drawing. We also define a setupBoard function which places the first four pieces on the board.

Making a move in Reversi is more complex than in the other games, since existing pieces must be reversed. To do this, we override the move function in protoBoard. Determining whether a move is valid is also more complex, since we require that the user flip some pieces. There are no "winning moves" per se. Rather, the game is scored when there are no more legal moves. We make this determination in makeComputerMove, and then echo it in winningMove and tieGame. Again, see the CD for the complete source code.

THE TAPBOARD APPLICATION TEMPLATE

While the functionality of making a move by either side is encapsulated in the game board templates and the protoBoard proto template, keeping track of whose turn it is and which game is being played is the responsibility of the application template. This template also provides Help and Credits buttons and coordinates activities when the application opens and closes.

The Your Move and Computer's Move radio buttons are simple protoRadioButton templates enclosed in the protoRadioCluster. After a user move is recorded, the Computer's Move button is turned on. The viewIdleScript of protoBoard notices this state and makes the computer's move, which in turn sets the Your Move radio button.

The game selection buttons are also protoRadioButton templates within a protoRadioCluster named gamePicker. When the value of these buttons changes, the gamePicker clusterChanged function starts the appropriate game by calling the newGame function in the application. The newGame function finds the appropriate game board and makes it visible; then it starts with the user's turn.

```
gamePicker.clusterChanged := func()
begin
   // Find the name of the new button.
   foreach t in stepChildren do
      if t.buttonValue = clusterValue then
         begin
            // Found it; start a new game with that name.
            :newGame(t.text);
            return;
         end
end

TapBoard.newGame := func(nm)
begin
   // Look through all the boards.
   local b;
   foreach b in boardList do
      // Are we looking for the one that's currently displayed?
      if nm = nil then
         begin
            // If so, and if this is it, clear it.
            if Visible(b) then b:clearBoard();
         end;
      // If not, check if this is the one that's been specified.
      else if StrEqual(b.name, nm) then
         begin
            // Set the current board, clear it, show it, and set the
            // piece icons.
            currentBoard := b;
            b:clearBoard();
            b:Show();
            player1Sample.icon := b.player1Piece;
            player1Sample:Dirty();
            player2Sample.icon := b.player2Piece;
            player2Sample:Dirty();
         end
      // If this isn't it, hide it (harmless if already hidden).
      else b:Hide();
   // Always start with the user's turn.
   :turn(kUserPiece);
end
```

As a shortcut, newGame uses boardList, an array that lists the available boards (one for each of the three games). It could have searched through the entire view list, but it's much quicker to have an array that lists just the game boards. This array, boardList, is created in the application's viewSetupFormScript and filled in by protoBoard's viewSetupFormScript.

```
TapBoard.viewSetupFormScript := func()
begin
   boardList := [];
   . . .
end

protoBoard.viewSetupFormScript := func()
begin
   // Register ourselves with the application.
   AddArraySlot(boardList, self);
   . . .
end
```

The newGame function also sets an application slot called currentBoard, which keeps track of the currently displayed board.

The two application-level functions below make it easy to set and find out whose turn it is. These functions are used within protoBoard and the game boards.

```
TapBoard.turn := func(p)
begin
   userOrComputer:SetClusterValue(p);
end

TapBoard.isTurn := func(p)
begin
   return p = userOrComputer.clusterValue;
end
```

DISPLAYING INFORMATION

The announceWin utility function brings up protoGlance templates (text views that appear for a brief time only) to announce game winning, losing, and tying. Other utility functions bring up and remove the "Working..." display. And finally, two utility functions bring up the help and credits protoFloatNGo templates (floating views with close boxes). All of these templates are in linked subviews.

```
TapBoard.announceWin := func(p)
begin
   // Make sure the current game display is up to date.
   RefreshViews();
   // Bring up the right glance view.
   if p = kUserPiece then youWin:Open()
   else if p = kNewtonPiece then iWin:Open()
   else tie:Open();
end

TapBoard.startWorking := func()
begin
   // Open the view.
   working:Open();
   // Force a refresh of anything that might need it; we're about to do
   // lots of time-consuming work, so the system won't get a chance to do
   // this otherwise.
   RefreshViews();
end

TapBoard.stopWorking := func()
begin
   working:Close();
end

TapBoard.announceHelp := func()
begin
   help:Open();
end

TapBoard.announceCredits := func()
begin
   credits:Open();
end
```

SAVING AND RESTORING THE STATE

When the application opens, viewSetupDoneScript checks whether there's a saved state in the System soup. If there is, it restores the state. When the application closes, viewQuitScript saves the state. A utility function, getStateEntry (described earlier), returns the current state entry in the System soup, if there is one.

```
TapBoard.viewSetupDoneScript := func()
begin
   // Find the saved state.
   local stateEntry := :getStateEntry();
   // Is there one?
   if stateEntry = nil then
      // If not, default to the first radio button.
      gamePicker:setClusterValue(1)
   else
      begin
         // If there is, restore the state from the entry.
         gamePicker:setByName(stateEntry.name);
         currentBoard:restoreState(stateEntry);
      end;
end

gamePicker.setByName := func(nm)
begin
   // Find the radio button with this name and set it.
   local t;
   foreach t in :ChildViewFrames() do
      if StrEqual(t.text,nm) then
         begin
            if clusterValue <> t.buttonValue then
               :setClusterValue(t.buttonValue);
            return;
         end
end

TapBoard.viewQuitScript := func()
begin
   // If any board is displayed (which it always will be -- we're just
   // being paranoid), save the current state.
   if currentBoard <> nil then currentBoard:saveState();
end
```

REMOVING THE SOUP ENTRY

When the application is removed from the Newton, it needs to remove the entry in the System soup so that it doesn't permanently waste Newton memory. This is done in the RemoveScript function. Note that this function is called both when the application is removed and when the card it's on is taken out of the Newton. We could distinguish these two cases, and not remove the soup entry if it's simply the card being pulled out, but we want to make sure we don't clutter up precious memory with game trivia if the card is never reinserted.

```
RemoveScript := func(packageFrame)
begin
   local cursor := Query(GetStores()[0]:GetSoup(ROM_SystemSoupName),
                         {type: 'index, indexPath: 'tag,
                          startKey: kPackageName, validTest: func(item)
                             StrEqual(item.tag, kPackageName)});
   if cursor:Entry() <> nil then
      EntryRemoveFromSoup(cursor:Entry());
end;
```

ALLOWING FOR OTHER SCREEN SIZES

We also need to consider the possibility that TapBoard may be used on a Newton with a screen size larger or smaller than that of the first Newton model (240 x 336 pixels). First we define all views in the bottom half of the application window to be relative to the bottom of the view, and views in the top half relative to the top. Leaving about 20 pixels of space between these two sets of views allows the application to shrink by that much, or to grow by a bit. We then need to dynamically set the bounds of the application to fit the screen, using the system function GetAppParams. If the screen is much larger than 240 x 336 we center it instead. All of this is accomplished in the application's viewSetupFormScript:

```
TapBoard.viewSetupFormScript := func()
begin
   . . .
   // Remember the original dimensions in case we need them.
   local originalWidth := viewBounds.right - viewBounds.left;
   local originalHeight := viewBounds.bottom - viewBounds.top;
   // Default the app bounds to the screen bounds (nice on small screens).
   local ap := GetAppParams();
   self.viewBounds := RelBounds(0, ap.appAreaTop,
                                ap.appAreaWidth, ap.appAreaHeight);
   // But if the screen's too large for that to look good, center it.
   // (We allow for a range of sizes to handle future screens.)
   if ap.appAreaWidth > (originalWidth+20) then
      self.viewBounds.right := originalWidth;
   if ap.appAreaHeight > (originalHeight+20) then
      begin
         self.viewBounds.top := ap.appAreaTop +
                                (ap.appAreaHeight - originalHeight) div 2;
         self.viewBounds.bottom := self.viewBounds.top + originalHeight;
      end;
end
```

SOME NEWTONSCRIPT EXERCISES

The following modifications to TapBoard would make good NewtonScript programming exercises:

- Replace the three linked subviews used by announceWin with a single template, with different text set programmatically within announceWin.
- Move the Your Move and Computer's Move radio buttons into the protoBoard proto template. Consider the ways this simplifies the application and the ways it complicates it.
- Add an elapsed time display showing how much total time the user and the Newton have taken to make their moves.
- Improve the algorithm for playing Tic-tac-toe.
- Add Go to the set of games.

SUMMING UP

As you've seen, TapBoard reduces code size by using proto templates to abstract out the redundant elements of the three games. This also reduces development time by making the application simpler, and easier to understand and modify.

The protoBoard proto template used in TapBoard is fairly large, implementing a substantial amount of functionality. Other uses of proto templates are much smaller, ranging from custom button types to modified versions of standard views or even completely new classes of templates.

Now you're ready to write some proto templates of your own — assuming you've ordered the Newton Toolkit from APDA. Be sure to share your protos with your friends!

THANKS TO OUR TECHNICAL REVIEWERS
Chris Christensen, Bob Ebert, Mike Engber,
Martin Gannholm, Kent Sandvik, Maurice Sharp,
Gregg Williams•

PETE ("LUKE") ALEXANDER

PRINT HINTS

TRACKING QUICKDRAW GX MESSAGES

In this column, I'd like to bring a tool called MessageWatcher to your attention. This tool, which is provided on this issue's CD, will help you understand the messages sent to your QuickDraw GX printer driver or printing extension when an application prints a document through the QuickDraw GX system.

A LITTLE BACKGROUND

develop Issue 15 gave an overview of QuickDraw GX in the article "Getting Started With QuickDraw GX" and discussed the QuickDraw GX messaging system in "Developing QuickDraw GX Printing Extensions." You can also learn about QuickDraw GX printing and messaging in *Inside Macintosh: QuickDraw GX Printing Extensions and Drivers*. If you've read these and don't need a refresher, you can just skip to the next section.

QuickDraw GX's extensible printing architecture makes writing printer drivers easier than ever. Easier yet is writing printing extensions, which allow you to modify the behavior of the QuickDraw GX printing system (similar to the way system extensions let you change the system by patching traps).

When an application prints through the QuickDraw GX system, QuickDraw GX either performs the requested task or sends a message (via the Message Manager) to the printer driver to perform the task. The QuickDraw GX printing system defines over 150 messages. For many tasks, QuickDraw GX provides a default implementation for the associated message, but sends a message to the driver anyway. The driver can then perform the task in its own way or massage the message parameters before forwarding the message on to the default implementation. In a printing extension, you can intercept and override any message before it gets to the printer driver.

ABOUT MESSAGEWATCHER AND OUR EXAMPLE

With over 150 messages in the QuickDraw GX printing system, it's a little difficult to follow the flow of the messages through the system or figure out their default implementation. MessageWatcher helps you with this by showing the messages being sent while an application is printing a document. This "live" view is very helpful toward understanding the calling chain and hierarchy of messages.

The MessageWatcher application and system extension are on this issue's CD. To use MessageWatcher, first drop the system extension into the System Folder and reboot; then launch the MessageWatcher application. Whenever any application initiates a print job, a window corresponding to that job will be opened in MessageWatcher and the messages sent as part of the job will scroll past. The title of each MessageWatcher window contains the internal ID of the job. (You should view the internal ID of a job similar to the way you view a file reference number today.)

The QuickDraw GX Finder printing extension (which is part of the QuickDraw GX system extension) and PrinterShare GX are treated like any other applications printing through the QuickDraw GX printing system. So they each have a print job associated with them, and MessageWatcher displays windows for them as well. If you were to create a desktop printer with the Chooser, there would also be a window for the Chooser.

As an example, we're going to print a document created and displayed by the "All Shapes with Printing" sample application, which is part of the sample code included with QuickDraw GX. This sample creates all of the shapes possible within the graphics system. Below is the code that prints the page of shapes; keep it in mind

PETE ("LUKE") ALEXANDER will tell you that life at Apple is hard. He spends long days (and frequent nights) with smart people trying to figure out what the right thing is and how to make it happen. Then he gets sent all over the world to tell people about what's new and happening. Last year Luke went to London and walked around the streets for ten hours trying to fend off jet lag and punk rockers, to Madrid where he had an unforgettable chocolate shake, to Milan where he saw the basement of the Apple building and tried not to take it personally that they wouldn't let him near a window, to Munich and Frankfurt where he finally understood why Germans gush about their beer, and to Sweden where he gave his QuickDraw GX talk in a discotheque filled with engineers who couldn't get the beat. After these whirlwind tours, Luke soars above the stress in his glider, but only occasionally takes an Apple Evangelist along, since they seem to think they've got connections up there and are forever telling him how to fly. Yes, life at Apple is hard, but after five years, Luke's beginning to get the hang of it.•

while going through the example of using MessageWatcher in the next section.

```
OSErr DoPrintOneCopy (WindowPtr myWindow)
{
   Str255   windowTitle;
   OSErr    printError = noErr;

   if (myWindow)
   {
      GetWTitle(myWindow, windowTitle);
      // Start sending the job. The job has the
      // same name as our window and contains
      // one page. This name appears in the
      // status messages sent to the desktop
      // printer.
      GXStartJob(gDocumentJob, windowTitle, 1);

      // Send the entire page of shapes to the
      // printer. (All the shapes being printed
      // have been collected into the GX picture
      // shape: gthePage.)
      GXPrintPage(gDocumentJob, 1,
               GXGetJobFormat(gDocumentJob, 1),
               gthePage);

      // Tell QuickDraw GX printing we're done
      // sending the job, so terminate the
      // spooling process.
      GXFinishJob(gDocumentJob);

      if (GXGetJobError(gDocumentJob) != noErr)
         // Your error-handling code here!
   }
}
```

We're going to look at our sample application printing to a LaserWriter II SC, so the message sequence displayed by MessageWatcher contains messages that are sent to a raster printer. Note that most of the messages would be different if we were printing to a PostScript® printer or to a plotter.

Each line in a MessageWatcher window consists of a message preceded by one of the following labels: send message, send object, and send object to. These labels represent the similarly named API calls in the Message Manager — for example, "send message" corresponds to the SendMessage call. All the "send" labels are roughly equivalent, so you'll be able to follow along well enough without distinguishing between them; if you do want more information about the Message Manager calls, see Chapter 6, "Message Manager," in *Inside Macintosh: QuickDraw GX Environment and Utilities*.

THE RETURN OF 'STR ' (–8192)

In the Print Hints column in Issue 13 of *develop*, I talked about the disappearance of 'STR ' (–8192) and 'PAPA' (–8192) resources from a system running QuickDraw GX and how your application should be prepared to handle this situation. The 'PAPA' (–8192) resource is still gone, but the 'STR ' (–8192) resource, containing the name of the default printer driver, has returned. Unfortunately, there are a few applications that require this resource to be present to run correctly — in some cases, to run at all. Since we wanted old applications to have the highest possible compatibility while printing through a QuickDraw GX system, we put the resource back. However, as mentioned in Issue 13, it's still not a good idea to depend on the information in 'STR ' (–8192).

While going through the various MessageWatcher windows in the next section, I won't bore you with a line-by-line discussion of everything displayed in each window. I'll give you the general idea behind the information presented; by checking out the documentation and trying out MessageWatcher yourself on different applications, you'll be able to figure out these details easily enough.

WHAT MESSAGEWATCHER DISPLAYS

The first time our sample application calls the QuickDraw GX printing API, the QuickDraw GX Finder printing extension initializes the default printer and then shuts down. The MessageWatcher window for this extension contains the following (the window title is shown in boldface; the ID displayed in each title will be different when you run MessageWatcher yourself):

```
Finder: 0x0000fac4
send object to: initialize
send object: defaultPrinter
send object to: shutDown
```

You'll also see PrinterShare GX start up a couple of times: The first time, it performs some general spool file and desktop printer queue management; it examines all the spool files and makes sure that the print jobs that are waiting are handled in the appropriate order. The second time it runs, it starts up the print job. The MessageWatcher window for PrinterShare GX displays this:

```
PrinterShare GX: 0x0000fa3c
send object to: initialize
send object: defaultPrinter
send object to: initialize
send object: defaultPrinter
send object: defaultPaperType
send object: defaultFormat
send object: defaultJob
   send object: defaultPaperType
send object to: shutDown
```

Notice that MessageWatcher indents some of information in the window, allowing you to see the message hierarchy. For example, the above window shows that the recipient of the defaultJob message sent the message in the indented line below it, defaultPaperType.

The next application to send messages through the system is our sample application, All Shapes with Printing. In this case, we chose Print One Copy from the File menu of the application to start the printing process. The application starts by setting up various default printing structures associated with the print job.

```
All Shapes with Printing: 0x00990c
send object to: initialize
send object: defaultPrinter
send object to: initialize
send object: defaultPrinter
send object: defaultPaperType
send object: defaultFormat
send object: defaultJob
   send object: defaultPaperType
```

Now we'll see the actual messages sent to print a document. Our sample application is ready to start up the print job and create the spool file. It uses GXPrintPage to print the document; you'll see the startJob and finishJob messages in the MessageWatcher window because GXPrintPage sends these messages in its default implementation. Next, the data contained on the page is spooled to disk. Finally, the print job is finished, and the spool file is closed and waiting for PrinterShare GX to find it and send it to the printer.

```
send object: startJob
   send message: createSpoolFile
   send object: jobStatus
send object: printPage
   send message: StartPage
      send object: jobStatus
   send message: finishPage
      send message: spoolPage
         send message: spoolData
         send message: spoolData
         send message: spoolData
         send message: spoolData
         send message: spoolData
send object: finishJob
   send message: completeSpoolFile
      send object: spoolResource
   send object: jobStatus
send object to: shutDown
```

The QuickDraw GX printing system is now ready to send the data contained in the spool file to the printer. PrinterShare GX starts by checking the status of the printer and then writing (that is, sending) the data. It continues to write the data and check the device status until all the data is sent. If a status check were to reveal that an error had occurred, the error would be available via GXGetJobError to the driver or extension (not to the application, which is usually out of the printing process at this point). Once all the data has been sent to the printer, PrinterShare GX checks the status of the printer one last time, makes sure the print job was successful, closes the spool file, and shuts down. We now have a piece of paper coming out of the printer.

```
PrinterShare GX: 0x00092520
send message: getDeviceStatus
send message: writeData
   send message: getDeviceStatus
      lots of writeData and getDeviceStatus
      messages to send the data to the printer
   send message: writeData
   send message: checkStatus
send object: jobStatus
send object: closeSpoolFile
send object to: shutDown
```

The MessageWatcher window for the QuickDraw GX Finder printing extension shows the messages it receives to update the status information being displayed in the desktop printer's window. In this case, the status is updated six times during the process of printing the document. To see exactly what status information is being sent to the desktop printer's window, you could open the window yourself and take a look. Remember, the messages will be slightly different for different types of printers — PostScript, raster (QuickDraw based), and plotter.

```
Finder: 0x000912dc
send object: writeStatusToDTPWindow
send object: writeStatusToDTPWindow
send object: writeStatusToDTPWindow
send object: writeStatusToDTPWindow
send object: writeStatusToDTPWindow
send object: writeStatusToDTPWindow
```

WATCH OUT NOW . . .

Following the flow of messages through the QuickDraw GX printing system will give you a better understanding of the calling chain and hierarchy of messages that are sent while a document is being printed. With MessageWatcher, you have the power to peek into the world of QuickDraw GX messaging. Happy exploring!

REFERENCES

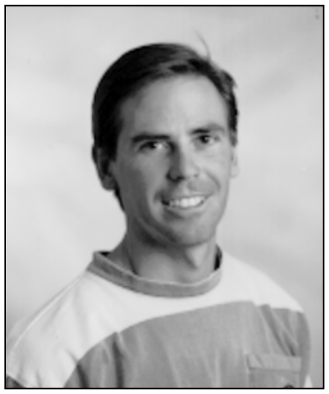
- *Inside Macintosh: QuickDraw GX Printer Drivers and Extensions* and *Inside Macintosh: QuickDraw GX Environment and Utilities.* On-line versions accompany QuickDraw GX releases; printed manuals will soon be available (Addison-Wesley, 1994).
- "Getting Started With QuickDraw GX" by Pete ("Luke") Alexander and "Developing QuickDraw GX Printing Extensions" by Sam Weiss, *develop* Issue 15.
- "Print Hints: Looking Ahead to QuickDraw GX" by Pete ("Luke") Alexander, *develop* Issue 13.

Buried deep within QuickDraw GX is an About box. The exercise of how to get to this box is left for the reader.•

Thanks to Hugo Ayala, Tom Dowdy, and Dave Hersey for reviewing this column.•

STANDALONE CODE ON POWERPC

A new format for standalone code in the PowerPC world brings increased functionality and easier implementation. You'll no doubt want to port existing code resources and write plug-ins for the new platform. Here you'll learn how to do both while also retaining or building in the ability to run the standalone code on the old 680x0 platform.


TIM NICHOLS

Standalone code is an important part of the Macintosh environment and will continue to be in the age of the PowerPC processor. Such code takes many different forms and serves many different purposes. It can serve as a definition function — such as an MDEF or a WDEF — for Macintosh system software, act as a dynamic extension to an application, or find other, more esoteric uses. In the PowerPC world, it can also be used to port time-critical portions of an application written in 680x0 code.

This article shows you how to develop and package standalone code modules to run in both the PowerPC and 680x0 worlds. We start by discussing the differences between standalone code in the two runtime environments. Then we go through the steps of compiling, linking, and packaging different types of standalone code, and calling it from within your application. We look at the following:

- how an application can support a plug-in that contains code in both the 680x0 and PowerPC formats, illustrated by preparing a plug-in sort algorithm for a simple application called SuperSort
- how to use a similar mechanism to port time-critical portions of an existing application to the PowerPC platform
- how to make an existing WDEF into a "fat" resource — one that will work in either a 680x0 or a PowerPC environment, depending on the machine executing the code

SuperSort, the plug-in, and the WDEF, along with their source code, are all on this issue's CD. All the code can run on either the 680x0 or the PowerPC platform, although you do need MPW to compile it.

TIM NICHOLS (Internet tim.nichols@3do.com) says the eight years he spent earning his bachelor's and master's degrees at UC Santa Barbara in between trips to the beach were the best years of his life. At Apple, he was a member of the PowerPC software team, where he developed some of the first PowerPC applications for demos and performance evaluation. He now works at 3DO in their ROM/OS group doing drivers and low-level system software. When not working, he plays softball and volleyball, fueling his activity with pizza and burritos.•

This article assumes that you know how to write a standalone code resource for the 680x0 platform and that you have a general grasp of PowerPC technology and runtime architecture.

THE STORY ON STANDALONE CODE

The format of standalone code has changed in the PowerPC world. Standalone code in the 680x0 world is packaged in resources such as WDEFs and INITs, with limited functionality and significant restrictions on their implementation. PowerPC standalone code, on the other hand, can be packaged as a resource or stored in the data fork of a file and enjoys a more flexible and powerful mechanism for managing global data and importing and exporting functions based on shared libraries.

STANDALONE CODE IN THE 680X0 WORLD

In the 680x0 world, developers can write two types of code: applications and standalone code. Applications have special privileges that aren't available to standalone code. Perhaps the most notable is the ability to easily access global and static data via the A5 world. The A5 register is maintained by the Process Manager for each application, to facilitate access to the QuickDraw global data as well as application global and static data. All references to global and static data by the application are made via the A5 register.

By contrast, standalone code resources have no A5 world and therefore don't have access to global or static data. This can limit the functionality of the code. There are mechanisms to get around this limitation, but they differ from one environment to the next. THINK C has a mechanism for using A4 as a pointer to global data for standalone code, while MPW uses special functions and macros to create a pseudo A5 world for the code resource. Both of these place a burden on developers by forcing them to set up and restore the appropriate registers before they can access their globals.

STANDALONE CODE IN THE POWERPC WORLD

In the PowerPC world, there's only one type of code, known as a *code fragment*. A code fragment is a collection of code and its corresponding data. Fragments can be packaged in a number of different kinds of containers. A PowerPC application consists of one or more code fragments packaged in the data fork of the application. Part of the Macintosh system software consists of code fragments packaged in the Macintosh ROM. Standalone code is really just another code fragment packaged in a resource or in the data fork of a file.

Whether standalone code is packaged in a resource or in the data fork depends on how it's being used. If you're writing a PowerPC version of an existing code resource such as a WDEF or an XCMD, the standalone code should be packaged in a resource, for purposes of compatibility. (The existing code only knows to look for

For more on standalone code in the 680x0 world, see the Macintosh Technical Note "Stand-Alone Code, *ad nauseam*" and the article "Another Take on Globals in Standalone Code" in *develop* Issue 12 •

For an overview of PowerPC technology and runtime architecture, see the article "Making the Leap to PowerPC" in *develop* Issue 16 and the soon-to-be-available *Inside Macintosh: PowerPC System Software*.•

code in resources of a specific type; for example, the Window Manager only looks for window definition functions in resources of type 'WDEF'.) If, on the other hand, you're developing a new standalone code module as a plug-in or to accelerate some part of your application, the standalone code should be stored in the data fork of your application or plug-in file to fully exploit the PowerPC runtime environment. Code can be loaded rapidly and efficiently from the data fork of a file without using a large memory footprint, thanks to the mechanism of file-mapped virtual memory.

Fragments can export symbols (code or data) by name to other fragments and can import symbols by name from other fragments. Each fragment contains an array of pointers known as the *table of contents* (TOC), which allows the fragment to share symbols with other fragments and is used to reference the fragment's own global and static data. Each entry in the TOC is a reference to either an imported symbol from another fragment or a static data item in the fragment itself. For example, suppose the code fragment Foo exports a procedure DoThis, contains a single global variable gMyGlobal, and imports a function DoThat from the shared library Bar. The TOC will contain an entry for each one of these symbols (DoThis, DoThat, gMyGlobal), and each entry will point to the address of the corresponding symbol, as shown in Figure 1.

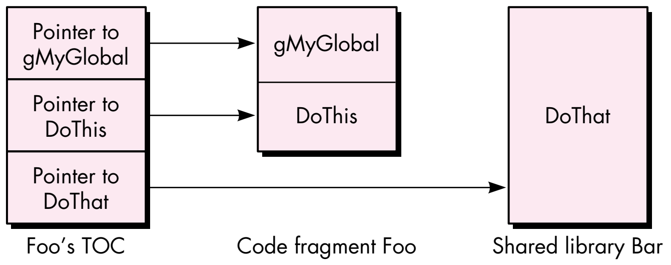


Figure 1
A Fragment's Table of Contents

The R2 register in the PowerPC processor is dedicated to storing the currently active TOC and thus is sometimes called the RTOC. The RTOC is saved, modified, and restored each time a new fragment is invoked. Because the TOC allows references to global and static data, it's analogous to the A5 world in the 680x0 environment. However, it's important to emphasize that in the 680x0 environment only applications have an A5 world and easy access to global and static data, while in the PowerPC environment, all fragments have a TOC and easy access to global and static data. So the great thing about standalone code being handled as a code fragment is that you can have globals in your WDEFs, INITs, and plug-ins without having to jump through any hoops at all!

Because a fragment can contain symbols from other fragments, these symbols must be resolved or bound at run time. This preparation is performed by the Code Fragment Manager. In most cases, such as when a PowerPC application is loaded, this is done transparently. Standalone code can be automatically prepared by the Mixed Mode Manager, but the preferred method is to have your application call the Code Fragment Manager directly. Fortunately, the Code Fragment Manager makes this an easy task, as we'll see later. Once a fragment has been prepared, the Code Fragment Manager returns a connection ID to identify the fragment. This connection ID is used when unloading the fragment, similar to a refNum that's returned when opening a file and later used to close the file.

The Code Fragment Manager has the ability to resolve symbols by name, so you can export any routine or data by name and then import that symbol in another fragment. This allows you to store multiple routines in your fragment, export them, and then call each routine when necessary by asking the Code Fragment Manager for its address. This is much nicer than having a dispatch-based, single-entry-point code resource as we do in the 680x0 environment.

CALLING STANDALONE CODE

At any given time a PowerPC processor–based Macintosh may be executing in the native PowerPC runtime architecture or in an emulated 680x0 runtime architecture. The switching between the two runtime environments is transparent and handled by the Mixed Mode Manager. And thanks to the Mixed Mode Manager, code from one instruction set can call code from another instruction set, which is just what happens when a 680x0 application calls a standalone code module written in PowerPC code or a native PowerPC application calls a standalone code module written in 680x0 code.

So whenever we're running on a PowerPC processor–based Macintosh and our application calls standalone code, we're presented with an interesting problem. Given a pointer to standalone code, how do we know what kind of code it points to? In the 680x0 world, a procedure pointer is simply the address of a procedure. But in the PowerPC environment, a procedure pointer is actually the address of a transition vector, which in turn contains pointers to the actual routine and the TOC for the fragment. Figure 2 shows the difference.

To solve this problem, the Mixed Mode Manager creates a generic procedure pointer known as a UniversalProcPtr (UPP). A UPP can point to one of two things: a 680x0 procedure (in which case the UPP is really just a 680x0 ProcPtr in disguise) or a routine descriptor (data type RoutineDescriptor). A routine descriptor is a data structure that describes the instruction set, parameters, and calling convention of the routine. The Mixed Mode Manager looks at the routine descriptor to determine whether a mode switch is necessary and, if so, how to perform the switch.

To run in a PowerPC environment, we use a UPP anywhere we would formerly have passed a ProcPtr, such as in specifying a dialog filter procedure. In the case of 680x0 standalone code (which typically is stored in a resource), we indirectly pass a ProcPtr, and thus a UPP, to the calling routine via the handle to the resource. For a PowerPC code resource (or for a "fat" resource), we have to replace this ProcPtr with a UPP, which points to a routine descriptor describing the routine in our code resource. Figure 3 compares the forms taken by the three different kinds of code resources (680x0, PowerPC, and fat).

Now that you have the necessary background information on standalone code, we can move on to demonstrate how to handle three different types of standalone code: a universal plug-in module, a module to port time-critical code, and a fat resource.

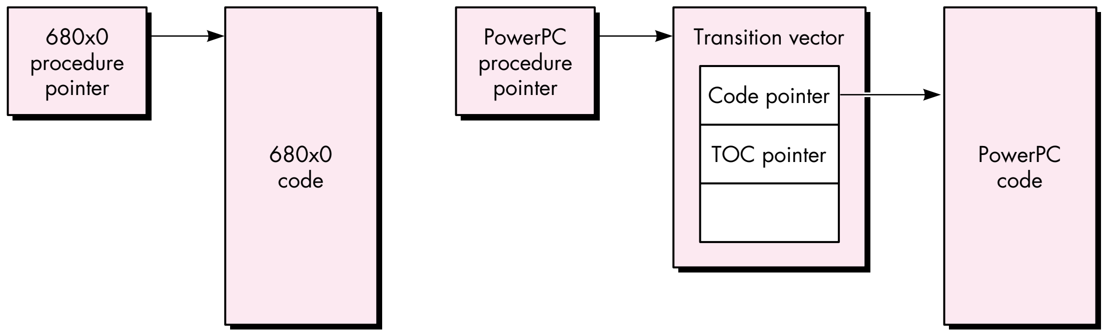


Figure 2
680x0 and PowerPC Procedure Pointers Compared

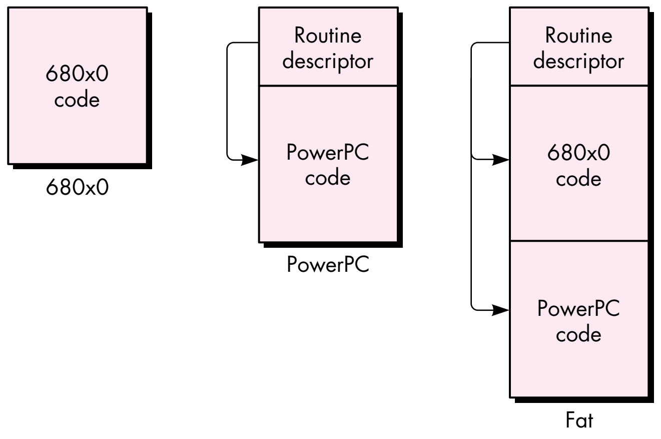


Figure 3
Forms of Code Resources Compared

A UNIVERSAL PLUG-IN MODULE

Plug-ins are a popular way for third-party developers to extend the functionality of an application. To demonstrate how to create and support a universal plug-in module — one that will run in either the PowerPC or the 680x0 world — we'll use the example of a plug-in module for an application called SuperSort, which you'll find on this issue's CD.

SuperSort is a simple application that visually sorts data represented as bars of varying height according to a specified algorithm. SuperSort has two built-in algorithms — bubble sort and quick sort — and can add new algorithms through a plug-in mechanism. We'll compile and package a shell-sort algorithm into a plug-in that will work with either the 680x0 or the PowerPC version of SuperSort. The application will pick the correct version of the plug-in automatically at run time.

EXAMPLE CODE

Below is the code for our plug-in sort routine that implements the shell-sort algorithm. ShellSort's data parameter is a pointer to the data to be sorted, the size is the number of elements to be sorted, and the swap parameter is a callback procedure to SuperSort to animate the sort.

```
#include "SortPlugIn.h"

#if powerc
#include <MixedMode.h>
ProcInfoType swapPI = kCStackBased
                           | STACK_ROUTINE_PARAMETER(1, kFourByteCode)
                           | STACK_ROUTINE_PARAMETER(2, kFourByteCode);
#endif

void ShellSort(DataPoint *data, short size, SwapProc swap);

void ShellSort(DataPoint *data, short size, SwapProc swap)
{
   short i, j, incr;

   incr = size / 2;
   while (incr > 0) {
      for (i=incr; i<size; i++) {
         j = i - incr;
         while (j >= 0) {
            if (data[j].n > data[j + incr].n) {
            #if powerc
               // We must use CallUniversalProc since we will be passed a
               // UPP for the SwapProc.
               CallUniversalProc(swap, swapPI, &data[j], &data[j + incr]);
```

For details on the shell-sort algorithm, see any book on algorithms, such as *Fundamentals of Computer Algorithms* by Horowitz and Sahni or *Data Structures and Algorithms* by Aho, Hopcroft, and Ullman.•

```
            #else
               // If we're 680x0, we can just call the proc directly.
               // MixedMode will handle switching if swap is a UPP.
               (*swap)(&data[j], &data[j + incr]);
            #endif
               j -= incr;
            } else
               j = -1;
         }
      }
      incr /= 2;
   }
}
```

COMPILING, LINKING, AND PACKAGING

We execute the following commands to compile and link this procedure in order to create the 680x0 version stored in a 'SORT' resource:

```
C ShellSort.c -o ShellSort.o
link -t 'rsrc' -c 'RSED' -m ShellSort -rt SORT=128 ShellSort.o     ∂
      -o ShellSort.rsrc
```

We compile and link the procedure again to create the PowerPC version to be stored in the data fork. The output of the PowerPC linker is known as an XCOFF (extended common object file format) file. This is a bloated file that we then strip to turn into a leaner file known as a PEF file (your guess as to what PEF stands for is as good as any). Here are the commands:

```
PPCC -w conformance -appleext on -sym full ShellSort.c -o ShellSort.c.o
PPCLink -main ShellSort -export ShellSort      ∂
      ShellSort.c.o                            ∂
      "{PPCLibraries}"StdCRuntime.o            ∂
      "{PPCLibraries}"PPCCRuntime.o            ∂
      -o ShellSort.xcoff
makepef ShellSort.xcoff  -e ShellSort          ∂
      -o ShellSort.pef
```

Now that we have the two pieces, we join them together:

```
duplicate -y -d ShellSort.pef  ShellSort
duplicate -y -r ShellSort.rsrc ShellSort
SetFile ShellSort -t 'SORT' -c 'TimN'
```

The resulting file, ShellSort, is our plug-in that can be executed on either the 680x0 or the PowerPC platform. The code fragment that's stored in the data fork will be loaded, prepared, executed, and unloaded by the PowerPC version of the SuperSort application, while the code contained in the 'SORT' resource will be loaded, executed, and unloaded by the 680x0 version.

CALLING THE PLUG-IN

When calling a universal plug-in, your native PowerPC application should first check to see whether there's a code fragment in the data fork of the plug-in, using the Code Fragment Manager routine GetDiskFragment. If so, the pointer returned by GetDiskFragment can be used to call the module. If not, the application should then look for the appropriate plug-in resource in the resource fork of the plug-in.

GetDiskFragment locates and loads a fragment found in the data fork of a file.

```
OSErr GetDiskFragment(FSSpecPtr fileSpec, long offset, long length,
         Str63 fragName, Mask findFlags, ConnectionID *connID,
         Ptr *mainAddr, Str255 errName);
```

The parameters are as follows:

fileSpec	The file to check for a fragment
offset	Offset into the data fork where the fragment resides
length	The length of the fragment, in bytes
fragName	The name of the fragment, used for debugging only
findFlags	The operation to be performed on the fragment
connID	The fragment connection ID
mainAddr	The main entry point of the fragment
errName	The error string returned if the call fails

Here's an example of how you might call a universal plug-in:

```
Handle         myProcHandle;
MyProcType     myProcPtr;
OSErr          err;
ConnectionID   connID;

err = GetDiskFragment(theFile, 0, 0, theFile.name, kLoadNewCopy,
            &connID, &myProcPtr, errName);
if (err == noErr) {
   /* We have a fragment, ladies and gentlemen! */
   (*myProcPtr)(p1, p2, p3);
   CloseConnection(connID);
} else
{
   /* We have a resource. */
   myProcHandle = Get1Resource(kMyCodeType, kMyCodeID);
   if (myProcHandle != nil) {
      HLock(myProcHandle);
      myProcPtr = (MyProcType)*myProcHandle;
      #if powerc
         CallUniversalProc(myProcPtr, kMyProcInfo, p1, p2, p3);
      #else
         (*myProcPtr)(p1, p2, p3);
      #endif
      HUnlock(myProcHandle);
      ReleaseResource(myProcHandle);
   }
}
```

The address that's returned is whatever symbol was defined as the main entry point during the linking of the PowerPC code. Because this is a true pointer to the routine and not a routine descriptor, it can be dereferenced and called directly as with any other ProcPtr you may be used to.

If your fragment has multiple entry points, you can use the Code Fragment Manager function FindSymbol after loading the fragment via GetDiskFragment in order to locate a particular symbol by name. The FindSymbol routine returns the address of the symbol you request.

A MODULE TO PORT TIME-CRITICAL CODE

To port only time-critical portions of your application, you would use a technique similar to the one just described. Factor out the code whose execution you want to accelerate, create a fragment, and package the fragment in the data fork of your application. In your application's initialization code, call the Code Fragment Manager to get the entry point to this fragment from your application's data fork and store this pointer. When you no longer need the pointer, call the Code Fragment Manager to close the connection to the code fragment.

Your code fragment will need a routine descriptor as its main entry point since it will be called from 680x0 code. To make a routine descriptor your main entry point, declare a global routine descriptor in your code that describes the fragment's main entry point. When you link the resulting object file, tell the linker to use this global routine descriptor rather than the actual code entry point as the main entry point.

Here's an example of using a global routine descriptor as an entry point to a fragment:

```
RoutineDescriptor MyEntryPointRD =
   BUILD_ROUTINE_DESCRIPTOR(kMyEntryPointProcInfo, MyEntryPoint)
```

When we go to link this code, we tell the linker that the main entry point is our routine descriptor.

```
link -main MyEntryPointRD -export MyEntryPointRD {MyObject} {MyLibs}   ∂
      -o {MyXCOFF}
```

A FAT RESOURCE

Although existing resources can run on a PowerPC processor–based Macintosh thanks to the 680x0 emulator, they run much more slowly than they would if they were written in native PowerPC code. If you make an existing resource “fat,” it will work in either the 680x0 or the PowerPC environment and you won’t need to ship two different versions of your resource. For example, if you have a fat WDEF, the code will run as usual on the 680x0 platform but will execute as native PowerPC code on the PowerPC platform, with the Macintosh system software choosing the correct code at run time.

CREATING A FAT RESOURCE

There’s a template defined in MixedMode.r that allows easy creation of fat resources. We’ll create a fat resource version of a WDEF to show how it’s done. We won’t present all of the code here but simply the steps involved in making the WDEF into a fat resource. The code for the WDEF is on this issue’s CD along with an application called TestWDEF that shows the WDEF working.

Recall from our earlier discussion that we call a code resource through a ProcPtr, which in this case is a dereferenced resource handle. That means that we need to create a routine descriptor for our PowerPC version of the WDEF so that the Mixed Mode Manager can invoke a mode switch, if necessary, when the system software calls the WDEF. This is consistent with the requirement that all ProcPtrs be replaced with UPPs in native PowerPC code.

Here’s an example of a fat resource Rez definition:

```
#include "MixedMode.r"

type 'WDEF' as 'sdes';

resource 'WDEF' (128) {
   0x00003BB0,                            // 680x0 ProcInfo
   0x00003BB0,                            // PowerPC ProcInfo
   $$Resource("WDEF.rsrc", 'oCod', 128),  // Name, type, ID of resource
                                          // containing 680x0 code
   $$Resource("WDEF.rsrc", 'pCod', 128)   // Name, type, ID of resource
                                          // containing PowerPC code
};
```

MixedMode.r is part of the Macintosh on RISC Software Developer’s Kit, soon to be available from APDA.•

The resource type 'sdes' is defined in MixedMode.r. The 'sdes' resource template inserts into the start of your resource some 680x0 code that checks whether you're running on a PowerPC platform. If so, it copies your PowerPC code to the start of the resource data in memory and calls the PowerPC code via a UniversalProcPtr embedded in the resource at the start of the PowerPC code. Once the PowerPC code has been copied, each subsequent call to the resource goes straight to the PowerPC code, bypassing the initial checks. If you're running on a 680x0 platform, the same process occurs, but instead the 680x0 code is copied over the resource data in memory. All of this is done transparently by the 'sdes' resource template.

CALLING THE FAT RESOURCE

If your fat resource was created using the template in MixedMode.r, you don't have to change your calling code to execute the PowerPC code fragment. Calling PowerPC standalone code is exactly the same as calling 680x0 code. Due to the magic of the fat resource, the calling code doesn't have to know the PowerPC processor even exists. It simply grabs the resource and calls it.

Here's what the code looks like:

```
Handle      myProcHandle;
MyProcType  myProcPtr;

myProcHandle = Get1Resource(kMyType, kMyID);
if (myProcHandle == nil) {
   // Handle the error.
   . . .
} else
{
   HLock(myProcHandle);
   myProcPtr = (MyProcType)*myProcHandle;
   (*myProcPtr)(/* Params go here. */);
   HUnlock(myProcHandle);
   ReleaseResource(myProcHandle);
}
```

When this code is compiled into 680x0 code, the parameters are placed on the stack and the actual routine is called via a 680x0 JSR(A0) instruction. When the JSR instruction is executed, the pointer in A0 points to a routine descriptor, not to 680x0 code. This causes the emulator to invoke the Mixed Mode Manager, which then performs the necessary context switch, automatically prepares the fragment for execution, and calls the PowerPC code. Upon exit from the PowerPC code, the Mixed Mode Manager performs a switch back to the emulated 680x0 environment and execution continues as if the call were to 680x0 code. The calling code never knows the difference.

NOW WHAT?

Now that you've learned the basics of standalone code on the PowerPC platform, you can start thinking about what you can do with your application or existing code resource to exploit the speed of the PowerPC processor. A good exercise is to consult your favorite algorithm book and create your own SuperSort plug-in using a different algorithm, or to recompile your favorite WDEF or other code resource into a fat resource that you can run on any Macintosh, whether PowerPC processor–based or 680x0-based. Remember to package your new code fragments in the data fork, and your recompilations of existing resources as resources. Then watch your creations take off!

REFERENCES

- "Making the Leap to PowerPC" by Dave Radcliffe, *develop* Issue 16.
- "Another Take on Globals in Standalone Code" by Keith Rollin, *develop* Issue 12.
- Macintosh Technical Note "Stand-Alone Code, *ad nauseam*" (Platforms & Tools 35).
- *Data Structures and Algorithms* by Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman (Addison-Wesley, 1983).
- *Fundamentals of Computer Algorithms* by Ellis Horowitz and Sartaj Sahni (Computer Science Press, 1978).
- *Inside Macintosh: PowerPC System Software* (Addison-Wesley, 1994).

THANKS TO OUR TECHNICAL REVIEWERS
Erik Eidt, Jim Gochee, Ed Navarrete, Jim Reekes•

THE VETERAN NEOPHYTE

WHY WE DO IT

DAVE JOHNSON

In the interest of pushing back the boundaries of human knowledge in my own small way, I decided to conduct a little survey via e-mail. I asked only one question:

> *What's so interesting about programming computers, anyway?*

I sent out 87 letters and received 37 direct responses, a 43% return rate! This is a fantastic return rate for a survey. As one respondent wrote: *What an interesting survey! Getting hackers to talk about themselves should be easy!*

It was.

To alleviate your suspense, here are the top two reasons programmers program, according to my survey:

1. Programmers like solving problems and puzzles. (49%)
2. Programming is a creative outlet. (36%)

In the survey itself I included some hypothetical answers to my own question, as follows:

> *Do you just like doing puzzles? Is it the money? Was it a horrible accident, and you were supposed to be a doctor when you grew up? Is it because you're really an artist, and computers are the best medium around? Is it because it's the only profession where you don't need to wear shoes to meetings? Is it because you get to hang out with smart people?*

Unfortunately, while this accurately conveyed my enthusiasm for the subject, it also completely shredded any objectivity my survey might have had, and skewed the results sharply toward the tight, square little box of my preconceptions. Note that the top reason appears in this list, and furthermore it appears first. Coincidence? I think not. As one alert respondent pointed out, I even blurred the important distinction between what makes programming interesting and what makes it a nice business to be in. Sheesh.

Abandoning rigor and embracing whimsy, I decided to dump all the responses into one big text file and do a word frequency analysis of the whole thing, logging how many times each word appeared. Ignoring the common words like *the* and the expected words like *program*, the winner was *problem*, at 12 occurrences. *Game* came in a close second, at 11 appearances. Three words tied for third place, at 8 occurrences: *fun*, *interesting*, and *think*. Now *this* is a satisfying analysis!

Other frequently appearing words were these: *puzzle*, *art*, *control*, *creation*, *enjoy*, *paid*, *challenge*, *complex*, *efficient*, *god*, *high*, *hooked*, *money*, *music*, *play*, and *solving*. Taken together, they make a strangely accurate mantra about what programming is like, and probably describe in a more complete way than any individual answer why people program.

But let's look again at those top two reasons. In second place we have "Programming is a creative outlet." That fact was mentioned over and over again, but creativity comes in many varieties, and it's interesting to look at the metaphors people chose to describe the act of writing software. Interestingly, they fell into three very distinct categories: programmer as mechanic, programmer as artist, and programmer as God.

The similarities between computer programming and mechanical engineering may be deeper than it seems at first glance:

DAVE JOHNSON's favorite book between the ages of 5 and 11 was the *World Book Encyclopedia.* Sometimes in the early mornings, before anyone else was awake, he'd sneak out of bed, creep quietly through the dark and silent house to the living room, and slip a volume from the shelf. ("M" is fondly remembered to this day: matches, monsters, motors . . .) Stopping off in the kitchen for a stack of cookies, he'd make his way back to his bed, and stay there reading and generating crumbs as the sun came up.•

I got hooked on erector sets when I was young.

— John Powers

But if programmers are just frustrated machinists, then why don't they get up from their computers and build real machines? Here's why:

On a computer, I can build an amazingly fantastic and elegant gadget, with thousands of working parts all cooperating pleasantly with one another, and I never have to get my hands dirty.

— Jeremy Nguyen

If you're willing to build your machines behind the glass of a computer monitor, to build them out of logical stuff instead of real stuff, then suddenly the landscape of possible machines becomes effectively infinite. You can build any machine you can *think* of, and all those pesky, annoying details like friction and inertia and tensile strength just fall away. For people who like to design machines, and especially for those who aren't so keen on the grease and skinned knuckles, this is a powerful attraction.

Other people chose to compare programming with creating art:

I'm not very good at art, but when I write something that works well I feel like an artist creating a piece of artwork.

— Kevin Mellander

My father is a painter, and I grew up watching him continually producing new artwork . . . Software is just another medium, less messy, easier to change.

— Fred Huxham

Programming is definitely a means of expression, and unlike other media software can simulate every *other* medium; in Alan Kay's words it's a *meta-medium.* For those with an artistic leaning this is rich ground, indeed: you can create your art *and* you can create new ways of creating art.

My own favorite creation metaphor is programmer as God:

I like it because you can play God, albeit on a microscopic scale. It's hard to explain, but writing code is like creating new life.

— Dave Hersey

It is the closest I can get to being God (without lifetimes of sadhana). I get to sculpt a little "life-form" that, while admittedly unsophisticated, has a satisfying degree of autonomy.

— Corey Vian

Since software is something that's dynamic in time and semi-autonomous, it can fairly convincingly imitate living things, in much the same way that machinery often can. But since the software "machines" that computers allow us to construct can be orders of magnitude more complex than machines in the real world, their resemblance to life can be much more compelling, much more striking.

The joy of solving problems (or the *compulsion* to solve them, as the case may be) seems to be the major reason people program. My favorite survey response of all sums this up nicely:

The reason I program is because I'm a compulsive problem solver and my computer is a never-ending source of problems.

— Craig Prouse

It's the twin meaning of "never-ending" that I love so much here. The fact that the problems never end is one of those "both blessing and curse" features that computers seem to overflow with. On the one hand, it means that we have an inexhaustible source of puzzles and problems to slake our thirsts, but on the other hand it means that we can never really be finished, and like Sisyphus rolling his stone up the hill, we are doomed to repetitive toil, solving the same kinds of puzzles over and over again.

Here are some of the other reasons people gave, which didn't fall neatly into any category:

> *It's basically because I have all these ideas in my head, and a computer is the best way to get them OUT of my head . . .*
>
> — Jeff Barbose

> *Solving the problems is fun, but I must admit that I enjoy the company of the machine — it's not every day you meet somebody who will associate with you unconditionally.*
>
> — Steve Beitzel

> *I program to get back at my senior year English teacher. I once got a bad grade on an English paper, and she couldn't give me any concrete reasons why. She just didn't like it. The computer, on the other hand, always gives me a reason why I'm incorrect. It's a game I play with my Mac . . . There are a bunch of rules and it tells me (usually via a bus error) that I broke one.*
>
> — Konstantin Othmer

> *It's the one thing in my life that is easy to debug.*
>
> — Michael Weingartner

These last two hint at something else that I think might be important. In sharp contrast to the real world, the world of the computer is reassuringly knowable. What this means is that *computer problems always have answers.* There is *always* a discoverable reason behind a computer's behavior, and I can't help wondering whether this is one of the major reasons people get such satisfaction from programming.

Most questions we ask about the real world turn out to be hopelessly complicated. How can we improve the economy? No neat and tidy answer to that one. Why does your child love those particular red plastic boots so much? That's not really answerable in any complete way. Why is ice slippery? Why is the sky blue? How do birds know when it's time to fly south? None of these questions have clean answers.

Ironically, though, there seems to be a basic human need to *find* answers. Look at the huge amount of time and energy we spend trying to discover and proclaim truths about our world. Religion, art, and science are all manifestations of the human desire to understand. So it's no surprise that computers, with their fantastically complex but ultimately knowable behavior, are extremely satisfying artifacts to spend time grappling with. Finding the answers can be enormously challenging, but there's *always* an answer to be found:

> *It gives me one thing I *know* I can boss around.*
>
> — Jim Luther

Amen.

RECOMMENDED READING

- *Dreams of Reason* by Heinz R. Pagels (Simon and Schuster, 1988).
- *The Hidden Life of Dogs* by Elizabeth Marshall Thomas (Houghton Mifflin, 1993).
- *Tuesday* by David Wiesner (Clarion Books, 1991).

Thanks to all the survey respondents and 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.•

DEBUGGING ON POWERPC

Debugging on a PowerPC processor–based Macintosh is just like debugging on any other Macintosh, only different. You should bring along the debugging skills you carefully honed on 680x0-based machines but expect the mechanics of debugging to be easier thanks to the PowerPC two-machine debugger. We give you basic instructions and provide a sample program that you can crash like crazy while you learn to debug PowerPC code.

DAVE FALKENBURG AND BRIAN TOPPING

The most important thing to realize when you set out to develop (and hence debug) for the PowerPC processor–based Macintosh is that this beast is still a Macintosh. Besides having a 680x0 emulator, the CPU has a Macintosh Toolbox in ROM, low-memory globals, a trap dispatcher, and 680x0 interrupt vectors. Since it retains so many elements you know and love, you don't have to throw away any of what you've learned about debugging with MacsBug, TMON, or any other debugger.

On the other hand, if all the new Macintosh had up its sleeve were 680x0 emulation, we wouldn't be writing this article. As described in "Making the Leap to PowerPC" in *develop* Issue 16 and in the imminent *Inside Macintosh: PowerPC System Software*, the PowerPC runtime architecture is new and improved. A couple of new managers — the Code Fragment Manager and the Mixed Mode Manager — help bridge the software gap between the 680x0 emulator and the PowerPC 601 microprocessor, and introduce some new twists and turns in how code is loaded and executed.

This article introduces you to the two-machine debugger developed for debugging PowerPC code. It then lays down some debugging ground rules, describes the circumstances in which your program might end up in the debugger, and discusses extensions and dcmds old and new to assist you in debugging. Finally, it talks about how to debug even in the absence of the debugger nub.

On this issue's CD you'll find CrashOMatic, a sample program you can use to explore the debugger without risking your own code. CrashOMatic is designed to cause

DAVE FALKENBURG (falken@apple.com on the Internet) begins his ideal day with French toast at Angelo's, continues with a #54 (poached chicken, Vermont cheddar, cucumbers, and ranch dressing grilled on challah) at Zingerman's, and ends at Metzger's with a large portion of brown food served with beer. Between culinary experiences he sandwiches in some work for the Macintosh Low-Level Toolbox Group at Apple. He just bought a house in California, so a trip back to Ann Arbor is probably out of the question. Dave has no pets with hair.•

crashes or demonstrate unusual aspects of PowerPC debugging. When launching CrashOMatic, hold down the Control key to force the debugger to take control.

To experiment with debugging CrashOMatic and to develop and debug native PowerPC applications, you'll need the Macintosh on RISC Software Developer's Kit (soon to be available from APDA) or one of the other PowerPC development kits available from third parties. The Macintosh on RISC Software Developer's Kit contains R2Db, an MPW-based cross-compiler called PPCC, and other assorted tools used for building PowerPC applications.

INTRODUCING R2DB

Apple's new debugger for the PowerPC processor–based Macintosh is called R2Db, for "RISC two-machine debugger." (As this issue goes to press, the fate of this name is undecided, so it may be different by the time you read this.) This modernized cross between ReAnimator and SourceBug allows for single stepping, setting breakpoints, and disassembling PowerPC code fragments. Like MacsBug, it's a systemwide low-level debugger; unlike MacsBug, R2Db also enables source-level debugging and is designed for debugging PowerPC applications. But as we hinted at earlier, you probably don't want to throw away your MacsBug skills just yet. R2Db can be used in conjunction with MacsBug, as explained in the section "Working With dcmds and MacsBug."

THE TWO-MACHINE SCHEME

R2Db is a two-machine debugger, as illustrated in Figure 1. The R2Db application runs on the *host machine*, which can be any Macintosh at all, preferably one with a large screen and enough CPU power to run a debugger built with MacApp 3.0. The part of R2Db called the PPC Debugger Nub runs on the *target machine*, the PowerPC processor–based Macintosh running the program you want to debug. The host machine acts as a remote control panel for the target. The machines need to be connected by a standard 8-pin printer cable.

Using two machines to debug code has several advantages. For one, a bug in your application that locks up the keyboard can't bring your debugging to a halt. For another, you can debug interrupt-level code without having to have incredible luck. (Can you say "MacsBug caused the exception"?) Throughout the development of Macintosh with PowerPC, the system software team relied on R2Db to debug such nasty (but necessary) things as the Memory and Resource Managers. Running a single-machine debugger on such shaky ground can lead to premature aging and the loss of some motor functions.

On the other hand, two-machine debugging has the disadvantage of requiring two Macintosh systems. (Oh darn, I guess it's time to ask the boss for another Macintosh Quadra.) Don't worry, though — Apple (and others) are busy working on single-machine debugging environments for those folks who develop on smaller budgets.

BRIAN TOPPING spends most of his days as part of the PowerPC team at Apple sandblasting the Memory Manager and debugging his Porsche 911. He's easily amused by such things as the kind of people who put Kleenex boxes in the back window of their cars and likes to exercise on weekends in mosh pits. He's currently as young as he'll ever be.•

To brush up on your debugging skills, see "Macintosh Debugging: A Weird Journey Into the Belly of the Beast" in *develop* Issue 8 and "Macintosh Debugging: The Belly of the Beast Revisited" in *develop* Issue 13.•

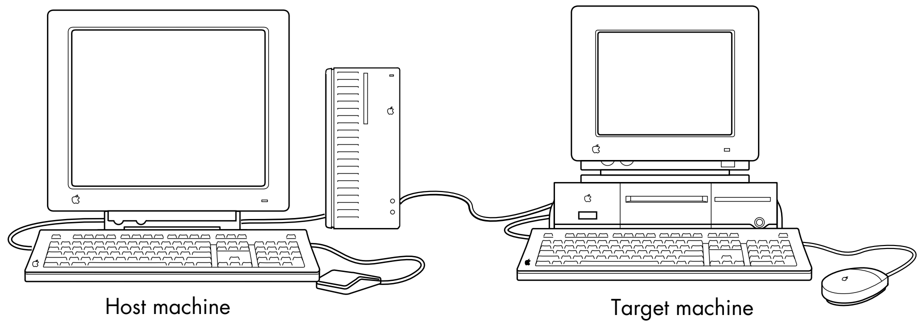

Host machine
(any Macintosh running R2Db)
Target machine
(Macintosh with PowerPC running the application being debugged)

Figure 1
The Two-Machine Debugging Scheme

R2DB BASICS

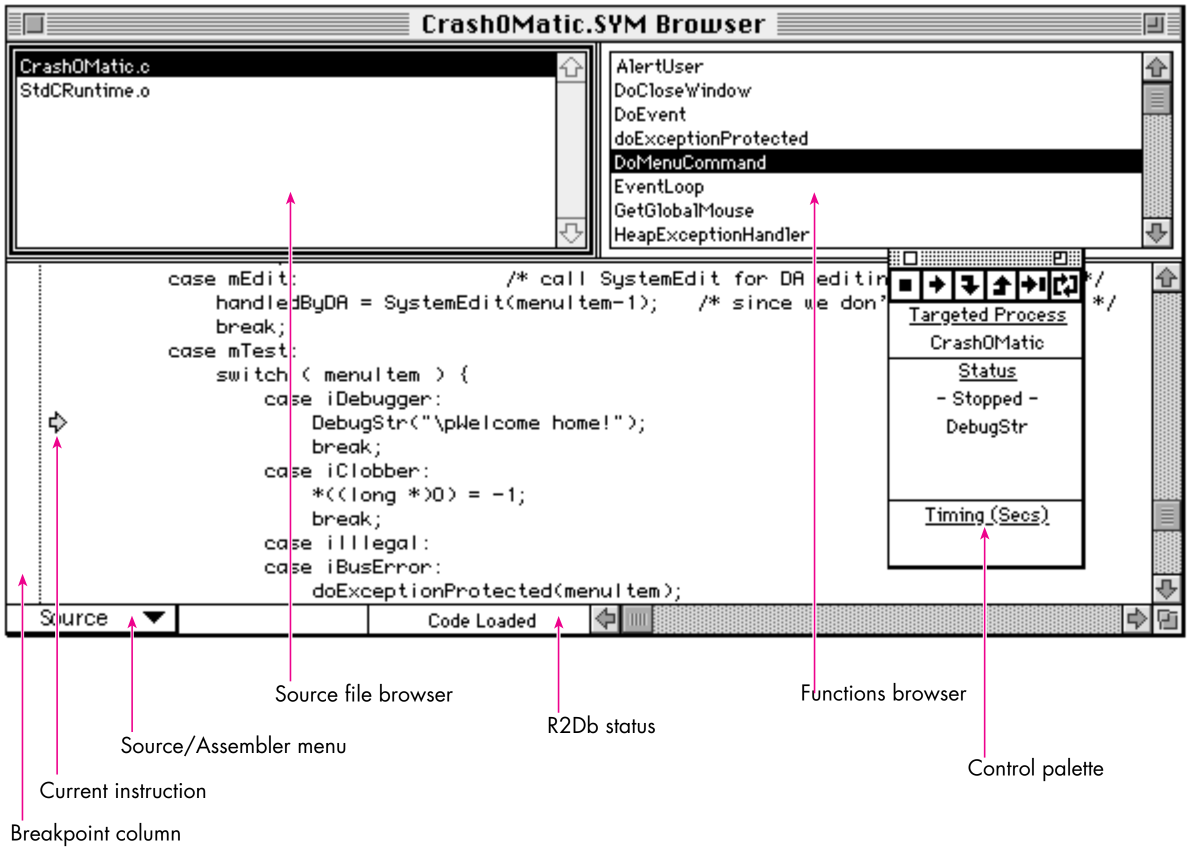
When you first launch R2Db, it presents a Standard File dialog box from which you choose an xSYM file to use when debugging. If you've used SourceBug, you know that a SYM file bundles together information about the application's source and object code and enables the debugger to associate a range of machine-language instructions with a line of C source code. The xSYM file is an extended version of the MPW SYM file that supports both 680x0 and PowerPC code. To support debugging "fat" applications — those with both 680x0 and PowerPC versions packaged together — two different kinds of SYM files are needed: the SYM file for the 680x0 version and the xSYM file for the PowerPC version.

When a PowerPC application is being debugged at the source level, its xSYM file and all its source code must be available on the machine running R2Db. Without the xSYM file, your application can still be debugged, but not at the source level.

When you choose an xSYM file, the R2Db browser window appears. This window, which will be familiar to users of Smalltalk or MacApp's Mouser, enables you to examine source code by file and function. Choosing Go To Debugger from the Debug menu makes the browser window look like the one shown in Figure 2. In the top left corner, a list of source files is presented. When a source file is chosen, the functions belonging to that file are listed in the top right corner. (Sorry, C++ fans — there's no object browsing in this release of R2Db.) The status of the target machine, along with a few stepping controls, appears in a control palette.

A small arrow points to the current instruction or line of code being executed. Breakpoints can be set by clicking to the left of the source display; a small hexagonal "stop sign" marks any breakpoints you've set. Double-clicking a breakpoint enables you to choose from a myriad of useful variants on the traditional behavior. Finally,

Don't try to use file sharing to make an xSYM file and its source code accessible from the target machine. When R2Db suspends the program being debugged, it will lock up the target machine, including file sharing.•



```
CrashOMatic.c
StdCRuntime.o
```

```
AlertUser
DoCloseWindow
DoEvent
doExceptionProtected
DoMenuCommand
EventLoop
GetGlobalMouse
HeapExceptionHandler
```

```
        case mEdit:                /* call SystemEdit for DA editin    */
            handledByDA = SystemEdit(menuItem-1);    /* since we don'   */
            break;
        case mTest:
            switch ( menuItem ) {
                case iDebugger:
                    DebugStr("\pWelcome home!");
                    break;
                case iClobber:
                    *((long *)0) = -1;
                    break;
                case illegal:
                case iBusError:
                    doExceptionProtected(menuItem);
```

Figure 2
The R2Db Browser During CrashOMatic Debugging

you can switch between assembly and source views by using the pop-up menu at the lower left of the display. To see exactly how the compiler translates your C code into native PowerPC code, select a line of source code and then switch to the assembly view, where it's highlighted.

Lurking under the R2Db menu bar are some useful commands, including commands to get register displays, memory dumps, and even 680x0 disassembly. If you ever find yourself stopped outside of your application, choose Show Instructions from the Views menu to get a disassembly at the current point of execution. This can also be helpful when debugging without an xSYM file.

R2DB IDIOSYNCRASIES

Like all things in life, R2Db has a couple of idiosyncrasies you should be aware of.

Debugger versus SysBreak. If you're familiar with SADE and SourceBug, you're probably accustomed to using SysBreak and SysBreakStr to add high-level breakpoints to your code. These functions aren't supported by R2Db, so you should use the familiar Debugger and DebugStr calls that you would normally use with MacsBug or other low-level debuggers. If a PowerPC debugger isn't installed, these calls are routed to MacsBug for handling. If you prefer using MacsBug to inspect data structures or log debugging messages, you can use the functions Debugger68k and DebugStr68k.

Memory Manager access faults. Starting with the Macintosh IIci, Apple added a hack to the existing Memory Manager to correctly support changes for 32-bit addressing and NuBus™ expansion cards. The change involved adding bus error wrappers within several internal routines to automatically call StripAddress and retry when a 24-bit handle is passed to the Memory Manager while the machine is temporarily operating in 32-bit mode. These bus error wrappers don't exist on 68000 machines like the Macintosh Plus, SE, and Classic (a fine reason to test your software on all sorts of machines).

These handlers also mask a serious problem: fake handles, fake pointers, and fake heap zones being passed to the Memory Manager. For Macintosh with PowerPC, the Memory Manager has been completely rewritten and actually preserves this tolerant behavior — but with a twist. When any PowerPC debugger is installed, these Memory Manager exceptions get routed through the debugger to point out the problem to the developer.

We would have added a "feature" to CrashOMatic so that you could see this behavior in action, but fortunately it doesn't happen that often. Before examining the fields of a handle or pointer block, the Memory Manager checks a magic cookie in the block as a first guard against fake handles. If we were to contrive an example without setting that magic cookie, the HLock call would still return an error code as it should; we just wouldn't see the bus error handlers get hit.

There are, however, some applications in which you will see the bus error handlers get hit. Open a file in ResEdit 2.1.1, for example, and you'll see access faults in R2Db. Choose Propagate Exception from R2Db's Control menu and the Memory Manager will clean up after ResEdit.

In future versions of Macintosh system software, this compatibility hack will be removed. Consider yourself warned.

GROUND RULES FOR DEBUGGING ON POWERPC

Before you start debugging your PowerPC application, you should commit the following ground rules to memory. This will ensure that you get off on the right foot.

For more about the evils of fake handles, see the Macintosh Technical Note "Memory Manager Compatibility" (Memory 13).•

Rule 1: Always use a nonoptimized build for source-level debugging. RISC C compilers radically reorder the sequence of instructions when generating optimized code. This makes straightforward source-level debugging impossible — imagine single stepping to the next line of code and having the arrow move to a statement three lines before where you just were. Following is a little program to demonstrate why you don't want to do source-level debugging with an optimized build of your code unless you really know what you're doing. Ignore the fact that "dude" is never initialized before being used.

```
void main(void)
{
   long  counter, dude;
   float fooVal = 1.0;

   for (counter = 1; counter < 1000; counter++)
   {
      fooVal = counter * fooVal;
      dude++;
   }
}
```

Below is the nonoptimized compiler output. (Incidentally, we compiled this program using the IBM compiler; if you compile it with PPCC you'll see different results.) Notice that the basic top-to-bottom structure of the C source is preserved.

```
Main:
   stw   r31,-4(SP)     ;# Preserve nonvolatile registers (r31)
   stwu  SP,-128(SP)    ;# Create stack frame

;# Variable initialization starts here.
   lwz   r31,xx(RTOC)   ;# Get address of a 1.0
   stw   r3,152(SP)     ;# Save r3 - we are going to use it
   lfs   fp1,0(r31)     ;# Put 1.0 in a register
   stfs  fp1,56(SP)     ;# Put 1.0 into fooVal

;# The FOR loop starts here.
   li    r3,1           ;# Get a 1
   stw   r3,60(SP)      ;# Put it in counter

   liu   r4,r0,0x4330   ;# Make floating-point version of counter
   stw   r4,96(SP)

   cmpi  cr1,r3,1000    ;# (counter < 1000)?
   bgt   cr1,Exit       ;# Goto end of loop
```

```
;# Body of the FOR loop starts here.
Loop:
   lwz   r3,60(SP)      ;# Load counter into register
   lfd   fp2,8(r31)     ;# Make fp_version_of_counter from counter
   xoris r3,r3,0x8000
   stw   r3,100(SP)
   lfd   fp1,96(SP)
   fsub  fp1,fp1,fp2
   frsp  fp2,fp1

   lfs   fp1,56(SP)     ;# Get fp_version_of_counter
   fmul  fp1,fp1,fp2    ;# fooVal = fooVal * fp_version_of_counter;
   frsp  fp1,fp1
   stfs  fp1,56(SP)

   lwz   r3,64(SP)      ;# dude++;
   addic r3,r3,1
   stw   r3,64(SP)

   lwz   r3,60(SP)      ;# counter++
   addic r3,r3,1
   stw   r3,60(SP)

;# Conditional test of FOR loop here.
   cmpi  cr1,r3,1000    ;# if (counter < 1000)
   blt   cr1,Loop       ;# goto loop

Exit:
   lwz   r31,124(SP)    ;# Restore saved registers (r31)
   addic SP,SP,128      ;# Release stack frame
   blr                  ;# Outta here!
```

The optimized compiler output is shown below. Note the interleaving of instructions used to initialize fooVal. Also note the radically different loop structure, which has no straightforward correspondence to the C code — counter is now zero-based and decrements, fooVal is calculated in a totally different fashion, and dude is nowhere to be found in the generated code.

```
Main:
   lwz   r3,xxx(RTOC)   ;# Get address of a 1.0
   li    r0,999         ;# counter = 999;
   lfd   fp1,8(r3)      ;# Finish fooVal = 1.0;

   fmr   fp0,fp1        ;# tmp = fooVal
   mtspr CTR,r0         ;# CTR = counter
```

```
Loop:
   fadd  fp0,fp0,fp1    ;# tmp = tmp + fooVal
   bdnz  loop           ;# CTR--; if (CTR != 0) goto loop
   blr                  ;# Outta here!
```

The moral of this story is that if you want to debug your application by looking at source code, you should create a special version compiled *without* optimization turned on. (Of course, the version you ship should be optimized.) If you can't cause the problem to occur with compiler optimizations turned off, you'll need to become familiar with debugging techniques involving animal or human sacrifice (or just learn to love reading optimized PowerPC assembly language).

Rule 2: Enable generation of symbol information by the compiler.
It's also important at build time to enable generation of symbol information by the compiler. This is analogous to "-sym on" and "-mbg on" for 680x0. The extra information generated is used by the MPW tools Link and MakeSYM to create the xSYM file that R2Db uses to enable source-level debugging. Here's a makefile to build a simple PowerPC program using PPCC (with debugging extras in boldface):

```
APPNAME     =  CrashOMatic
APPOBJECTS  =  CrashOMatic.o
PPCC        =  PPCC
PPCCOPTIONS =  -w conformance -appleext on -sym on -opt off

PEFOPTIONS  =  -ft 'APPL' -fc 'GDed'
LIBEQUATES  =  -l InterfaceLib.xcoff=InterfaceLib  ∂
               -l StdCLib.xcoff=StdCLib            ∂
               -l MathLib.xcoff=MathLib

{APPNAME}      ƒƒ {APPOBJECTS}
   PPCLink -warn  -sym on                          ∂
      {APPOBJECTS}                                 ∂
      "{PPCLibraries}"InterfaceLib.xcoff           ∂
      "{PPCLibraries}"StdCLib.xcoff                ∂
      "{PPCLibraries}"MathLib.xcoff                ∂
      "{PPCLibraries}"StdCRuntime.o                ∂
      "{PPCLibraries}"PPCCRuntime.o                ∂
      -o {APPNAME}.xcoff
   #  Create PEF executable from linker output
   makepef {APPNAME}.xcoff -o {APPNAME} {LIBEQUATES} {PEFOPTIONS}
   #  Rez in 'cfrg' (0) resource
   rez {APPNAME}.r -a -o {APPNAME}
   #  Create xSYM file for debugging
   makesym -o {APPNAME}.xSYM  {APPNAME}.xcoff
```

If you use the IBM AIX PowerPC compiler, check your documentation to find out how to disable optimization, suppress traceback information, set up structure alignment, and disable generation of code that uses the MQ register.•

```
CrashOMatic.o     ƒ  CrashOMatic.c
   {PPCC}  {PPCCOPTIONS}   CrashOMatic.c -o CrashOMatic.o
```

Rule 3: Always test with virtual memory both on and off.
Macintosh developers have long been used to having free access to any bits in memory, whether they contain code, data, or teachings of the illuminati. Meanwhile, users have been complaining that the Macintosh doesn't offer "modern" features like protected memory. The first release of System 7 for the PowerPC microprocessor includes the ability to protect PowerPC application code from errant write instructions; however, write protection is enabled only when virtual memory is turned on.

In addition, several PowerPC C compilers create string constants and initialized arrays in read-only sections by default. Modifying these values from your program will cause an access fault. For instance, it's surprisingly easy to write the following:

```
DebugStr(C2PStr("Hello, world"));
```

Because C2PStr modifies a string in place, your program will attempt to write into the read-only storage where "Hello, world" lives. There are compiler options to turn this feature off, but we don't recommend them.

The stricter runtime environment isn't the only reason to make sure that your software works properly when virtual memory is active. PowerPC programs are usually larger than their 680x0 cousins, so the likelihood of a user's enabling virtual memory on a PowerPC machine is much higher than on an earlier Macintosh. Apple has been telling you to be virtual-memory compatible for years. Just do it now and your customers will thank all of us later when Apple can release a protected-memory, preemptive multitasking version of the Macintosh operating system without breaking any of their favorite applications.

Rule 4: Don't write self-modifying code.
Unlike its later siblings the 603 and the 604, the PowerPC 601 has a merged data and instruction cache, which makes it much easier to write self-modifying code without getting caught. Remember all those 68040 compatibility problems you had when the Macintosh Quadra first came out? Don't forget what you learned. Also remember that writing to application code will cause an access fault in any PowerPC application when virtual memory is active.

Use the Code Fragment Manager to load all executable code. The Code Fragment Manager will take care of invalidating instruction caches (and flushing data caches) in an efficient manner on later versions of the PowerPC chip. If you're one of those crazed folks who still wants to write "structs" of code, custom-compiled shape blitters, or stub defprocs, be forewarned that you do so at your own risk.

PowerPC code is bigger than the equivalent 680x0 code because all instructions are four bytes long to make things easier for the hardware. Instructions are also typically register-based and may require a few surrounding instructions to accomplish the same task as a single 680x0 instruction. On the positive side, RISC compilers are much better at keeping these extra instructions to a minimum.•

With the ground rules laid, it's time to look at the reasons why your application might end up in the debugger.

EXCEPTIONAL CIRCUMSTANCES

Just like the 680x0, the PowerPC microprocessor has a list of things it can't handle without the help of developers like you. Any program can come to a screeching halt as a result of any of the following exceptions. (Note that these are hardware exceptions, as opposed to the software exceptions discussed in the article "Living in an Exceptional World" in *develop* Issue 11.)

`illegalInstructionException`

You executed some code that wasn't code. This usually happens when you accidentally call 680x0 code without using CallUniversalProc or one of its macro shortcuts.

`trapException`

A trap instruction that the debugger didn't know about was encountered. The most likely cause is hitting a developer-inserted debug trap.

`accessException`

This PowerPC version of a bus error usually occurs when a memory access was made in never-never land. You might encounter the fabled address 0xDEADBEEF in an access exception; someone (probably the same person who came up with the eieio instruction) decided it would be cool to initialize registers with this weird value to help you understand that you probably used an uninitialized variable.

`readOnlyMemoryException`

As mentioned earlier, when virtual memory is active, application code is mapped read-only for your protection. You may also see this exception when errant PowerPC code attempts to write to ROM. Sometimes this exception masquerades as a generic access exception.

`privilegeViolationException`

Remember how users wanted protected memory? Part of ensuring this capability in a future Macintosh is ensuring that nobody writes code that can mess with the operating system behind its back. Learn to live with it — the users who pay you want things this way. For a list of privileged instructions that you should avoid in your application, check out the *PowerPC 601 RISC Microprocessor User's Manual.* (By the way, even the debugger and the 680x0 emulator are written using only user-level instructions.)

Those interested in obscure hexadecimal numbers will want to know that 0x7F800008 is the value corresponding to one of the many variants of the PowerPC trap instruction.•

`traceException`

If the debugger goes astray, you may see this exception. It's pretty tough to cause.

In a departure from 680x0 Macintosh applications, PowerPC applications can attempt to field these exceptions before the dreaded bomb is emblazoned on the user's screen, without resorting to low-memory antics. Using the Exception Manager (new on the Macintosh with PowerPC) it's possible to catch memory faults, illegal instructions, and other faults within your application.

If you encounter an exception inside R2Db and would like to give the application a crack at fixing things, you can choose the Propagate Exception command from the Control menu. With some work, it's even possible to debug your application's exception handler.

USING OLD TRICKS IN THE NEW WORLD

Experienced developers know that extensions (such as Double Trouble and Dispose Resource) and dcmds (such as **rd**, **file**, **drive**, and **driver**) make the job of debugging go much more quickly. Most of the existing MacsBug extensions and dcmds work in the PowerPC world much as you would expect, but you should be aware of a few caveats.

WORKING WITH DEBUGGING EXTENSIONS

Tried-and-true debugging extensions install 680x0 code that will be emulated. When they detect a problem, they may behave differently depending on whether they're discipline-style extensions or memory-modification extensions.

Discipline-style extensions are extensions that patch traps to check parameters to calls for validity. Since the code that patches into the trap is emulated and signals failure via a 680x0 DebugStr trap, MacsBug is entered during a failure.

Memory-modification extensions such as EvenBetterBusError work by causing a bus error (also known as an access fault) in the problem program. You should remember that EvenBetterBusError works by setting the value at location 0 to be an illegal instruction, an illegal address, and an odd address, all in one 4-byte value. This catches lots of programs that accidentally use stale data in empty handles and nil pointers returned by NewPtr.

Through the magic of emulation, EvenBetterBusError works as before for 680x0 applications. Because EvenBetterBusError causes the problem to surface within the application (and not MacsBug), the bus error exception is thrown to the application's exception handler. If no exception handler is installed, control is passed to the PPC Debugger Nub. As with all PowerPC exceptions, if the debugger nub isn't installed, the PowerPC system software generates a 68000 "spurious interrupt" exception,

See the Exception Manager chapter of *Inside Macintosh: PowerPC System Software* for details on teaching your application to deal with exceptions.•

which is caught by MacsBug. We'll come back to this in the section "Debugging Without the Debugger Nub."

WORKING WITH DCMDS AND MACSBUG

Because R2Db does have its shortcomings (mostly due to lack of maturity), it gives you a way to enter MacsBug — by choosing Enter MacsBug from the Extras menu. This enables you to use almost any Macintosh debugging trick in the PowerPC world. You might want to do this, for instance, to gain access to commands that display data in forms not yet available in R2Db. For more information on how the old and new worlds coexist in a compatible yet forward-thinking manner, see "Traps and the PowerPC InterfaceLib: More Than You Want to Know."

When in MacsBug, you need to remember that you got there as a result of a Mixed Mode transition to a 680x0 R2Db subroutine that contains a 680x0 DebugStr, and nothing more. You can look and touch, but you can't step. Stepping will only walk you into the hands of the Mixed Mode monster, returning control to the PPC Debugger Nub and upward to R2Db. It's more dignified to return to R2Db when you choose to, by typing "G."

Be aware that the step-and-check commands such as **step spy** work only while the emulator is active. This is because they rely on 680x0 trace vectors or 680x0 breakpoints. The emulator is good enough to oblige MacsBug when between emulated instructions, but remember that the PowerPC microprocessor pays no attention to such things. This changes the definition of these commands from "do *neato thing you really like* after each instruction" to "do *neato thing you really like* after each 680x0 instruction." The difference is subtle but important, as illustrated by the following example.

Choose Step Spy from the Debug menu in CrashOMatic; this will bring up a dialog where you can type an address to spy on. Then choose Clobber from the Debug menu, which will bring up a dialog in which you type the same address. MacsBug will know that the memory got hit, but it won't know who hit it. Since the subroutine that did the clobbering and everything else back out to the main event loop is PowerPC code, the **step spy** won't get hit until the next call to the emulator (usually a Toolbox trap), in this case WaitNextEvent. You may be thinking, "Funny, it looks like there's no way that the previous instruction could have done that much damage," but remember that a lot of PowerPC code could have executed in the meantime.

Another funny thing about the 680x0 emulator is the way in which trap dispatching is performed for emulated code. To gain an immense performance boost, if the emulator recognizes that the A-line exception vector points at the ROM trap dispatcher, it takes some shortcuts and runs its own superfast version. Unfortunately, doing an **atb** in MacsBug modifies this behavior and throws the emulator into low gear. Just for fun, try doing an **atb** _Chain (a trap that never gets called) and see how much your machine slows down.

You can port MacsBug dcmds to the PowerPC platform very easily. For more information, check out the R2Db documentation that accompanies the Macintosh on RISC Software Developer's Kit.•

When you use MacsBug from within an R2Db session, sometimes the connection between host and target can be dropped. The connection can usually be reestablished by relaunching R2Db on the host machine.•

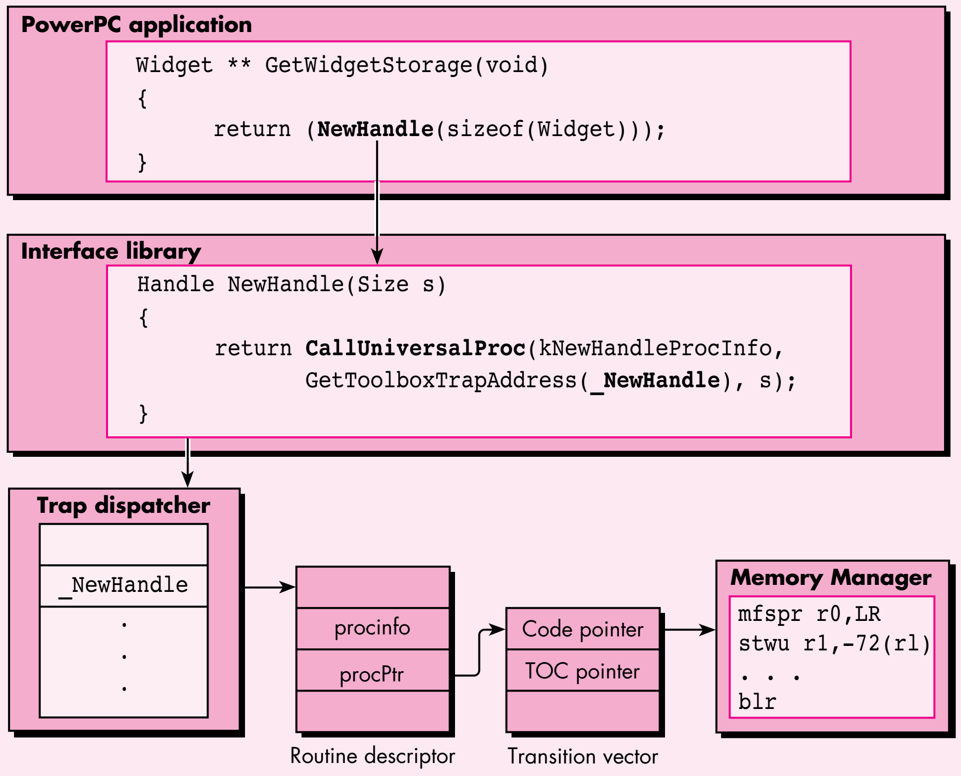
TRAPS AND THE POWERPC INTERFACELIB: MORE THAN YOU WANT TO KNOW

Why would you want to use MacsBug and R2Db at the same time? As mentioned earlier, Macintosh with PowerPC still has the familiar A-line trap dispatcher present on earlier Macintosh models. With the advent of system-supported shared libraries, it may seem strange to still support the trap dispatcher. Two main reasons motivated the use of the trap table by the PowerPC native toolbox: support should be provided for systemwide patching to preserve compatibility with existing System 7.x–friendly system extensions; and existing 680x0 software should be able to access PowerPC-accelerated Toolbox managers like QuickDraw.

Toolbox accelerations such as PowerPC QuickDraw are implemented by using NSetTrapAddress to patch out the 680x0 implementation with a PowerPC subroutine wrapped by a Mixed Mode routine descriptor. This allows existing 680x0 software to call PowerPC code in an application-transparent fashion.

On the other side of the coin, PowerPC applications must be able to call patchable versions of the Macintosh Toolbox. This is accomplished by building an interface library that calls through the trap dispatch table via CallUniversalProc, as illustrated below.

Understanding the behavior shown in the illustration is useful for debugging now, but in the future several new managers will be released that don't use this bizarre mechanism (most notably QuickDraw GX and the Thread Manager). For future compatibility, don't rely on this behavior.



```
Widget ** GetWidgetStorage(void)
{
      return (NewHandle(sizeof(Widget)));
}
```

```
Handle NewHandle(Size s)
{
      return CallUniversalProc(kNewHandleProcInfo,
             GetToolboxTrapAddress(_NewHandle), s);
}
```

```
mfspr r0,LR
stwu r1,-72(r1)
. . .
blr
```

Speaking of **atb**, the way PowerPC code calls Macintosh traps makes this dcmd almost useless for PowerPC debugging. Remember that **atb** works by replacing the ROM trap dispatcher with its own version, but because all PowerPC calls to the Toolbox are invoked via Mixed Mode and not the 680x0 emulator trap dispatcher, the new dispatcher is never invoked. Unfortunately, the same is true for **atr**. Fortunately, there is a fix for this in the form of a new dcmd.

NEW EXTENSIONS AND DCMDS

So far we've looked only at extensions and MacsBug dcmds created for the 680x0 environment. As you'd expect, many more have been created for debugging the changes that came with the PowerPC technology. In this section we look at some of the most important ones. These dcmds were created by the PowerPC development team to aid in the construction of Macintosh with PowerPC and are provided on this issue's CD in a completely unsupported and barely documented fashion. If they weren't so useful, the people in charge probably wouldn't let us give them to you.

atbv

Given a trap name, **atbv** sets a breakpoint on the trap vector so that both emulated and PowerPC callers can be intercepted by the debugger. Unlike **atb**, it doesn't ignore PowerPC callers or affect the emulator's ability to run at full speed.

brp

The dcmd **brp** stands for "breakpoint PowerPC." Given an address, **brp** will set a one-time breakpoint at the address in R2Db. Unlike **br**, its 680x0 cousin, the breakpoint set by **brp** is cleared once it's been hit.

cfm

All code must be loaded by the Code Fragment Manager before it can be executed. The dcmd **cfm** displays the Code Fragment Manager contexts of loaded PowerPC code and is thus a very powerful way to find exactly where an address lives. This dcmd is case sensitive.

Typing "cfm" all by itself gives a dump that looks like this:

```
CFM Info for all loaded fragments:
  contextID = 3: heapZone = 004cbe00  processName = "DiaTim"
   connID=3: 'DiaTim'#0; file (v=-1,d=232) "DiaTim" @ 0:#12592
      inf = 'peff','pwpc', sym = #0, use = 1, pef = 004cddf0, flg = 20
      sect 0:  @ 004ce3a0-004d0e78, exec, use = 1, len = #10968
      sect 1:  @ 004d36e0-004d6240, writ, use = 1, len = #11104
   connID=4: 'InterfaceLib'#0; inMem @ 409f3690:#214592
      inf = 'peff','pwpc', sym = #2466, use = 1, pef = 409f3690, flg = 00
      sect 0:  @ 40a03ac0-40a268ec, exec, use = 2, len = #142892
      sect 1:  @ 000143c0-0001b744, writ, use = 2, len = #29572
```

The entire dump shows many more libraries. All loaded code fragments, including shared libraries and applications, are listed. Every application running, whether emulated or PowerPC code, is provided with a Code Fragment Manager context ID. Individual code fragments, which include the application itself and any referenced shared libraries, are loaded into this context. The listing above shows that the application DiaTim has a code fragment context ID of 3, has a single executable (exec) code section and a writable (writ) data section, and references a shared library called InterfaceLib.

Typing "cfm" followed by a name will dump the code fragment with that name, plus the libraries it references. Remember that PowerPC applications are fragments, so this is quite useful for seeing only the fragments associated with your program. (In this example and the others that follow, what you type is shown in italic.)

```
cfm DiaTim
CFM Info for fragment named "DiaTim"
  contextID = 3: heapZone = 004cbe00  processName = "DiaTim"
   connID=3: 'DiaTim'#0; file (v=-1,d=232) "DiaTim" @ 0:#12592
      inf = 'peff','pwpc', sym = #0, use = 1, pef = 004cddf0, flg = 20
      sect 0:  @ 004ce3a0-004d0e78, exec, use = 1, len = #10968
      sect 1:  @ 004d36e0-004d6240, writ, use = 1, len = #11104
```

dis

The dcmd **dis** is like **il** for PowerPC instructions. Given an address that points to code, **dis** disassembles the PowerPC instructions at that address.

```
dis 409cdf28
   409cdf28   mfspr      r0,LR                          | 7c0802a6
   409cdf2c   stwu       SP,0xffffffc0(SP)              | 9421ffc0
   409cdf30   stw        r0,0x48(SP)                    | 90010048
   409cdf34   ori        r6,r3,0x0000                   | 60660000
   409cdf38   addis      r4,r0,0x0003                   | 3c800003
   409cdf3c   addis      r0,r0,0x0001                   | 3c000001
   409cdf40   addic      r4,r4,0x3932                   | 30843932
```

drd and pp

The dcmds **drd** and **pp** are used to examine the contents of a Mixed Mode routine descriptor. The **drd** dcmd, which stands for "display routine descriptor," is used to examine how a given trap is patched with PowerPC code.

```
drd NewHandle
drd: 00064ad0
  MixedModeMagic: 0xAAFE, version: #7, flags: 0x00 (NotIndexable)
  LoadLoc: 0x00000000, reserved2: 0x00000000, SelectorInfo: 0x00
       (No Selector)
  Routine Count (zero-based): 0x0000 (#0)
  ---- Routine Record 0x0000 (#0) at 0x00064adc ----
       ProcInfo: 0x00033132, Reserved1: 0x00000000, ISA: #1 (PowerPC)
       Record Flags: 0x0004 (Absolute, IsPrepared, NativeISA,
             PassSelector, IsNotDefault)
       ProcPtr: 0x00064458, offset: 0x00000000, selector: 0x00000000
```

Included in this information about the routine descriptor is the procInfo value, which describes the calling conventions, and the procPtr, which is the address of a transition vector. Using **pp**, which stands for "parse procInfo," we can convert a procInfo value into a more readable form.

```
pp 33132
ProcInfo: 00033132
  -----------------------------
  Calling Convention: 0x02 (#2) Register Based
  Return value:  4 Bytes in Register A0
  Parameter 1: 2 Bytes in Register D1
  Parameter 2: 4 Bytes in Register D0
```

Finally, we can use our old friend **dis** to look at the code, remembering that a PowerPC procPtr is a pointer to a pointer to code (hence the extra caret in the following).

```
dis 64458^
   40a9cb24   mfspr      r0,LR                           | 7c0802a6
   40a9cb28   stmw       r26,0xffffffe8(SP)              | bf41ffe8
   40a9cb2c   stw        r0,0x8(SP)                      | 90010008
   40a9cb30   stwu       SP,0xfffffe90(SP)               | 9421fe90
   40a9cb34   lwz        r30,0x0(TOC)                    | 83c20000
   40a9cb38   rlwinm     r29,r3,0,16,31                  | 547d043e
   40a9cb3c   addic      r6,SP,0x004c                    | 30c1004c
```

findsym

Typing "findsym" followed by a symbol name gives specific information about an exported symbol.

In the following example, we see that NewHandle lives in ROM at address 40a0d5d0 (your results will vary), has a TOC value of 1b704, and occupies the shared library called InterfaceLib.

```
findsym NewHandle

findsym: "NewHandle"
  "NewHandle" #952 TVec 00015f10 (40a0d5d0,0001b704) in "InterfaceLib" (1,4)
```

frown

The dcmd **frown**, which stands for "fragment ownership," is similar to **wh** in MacsBug but is used to display the code fragment and closest exported routine name associated with a given address. Unlike **wh**, **frown** doesn't give you any information about where an address is located within a Memory Manager heap.

```
frown 40a0d5d0

frown: 40a0d5d0
     is owned by: section #0 (exec,non-writ) of "InterfaceLib"
     is near: "NewHandle" #952 TVec 00015f10 (40a0d5d0,0001b704)
```

r2db

The dcmd **r2db** allows you to enter R2Db from MacsBug. It's the complement of R2Db's Enter MacsBug command. But because Enter MacsBug executes a _Debugger trap, it isn't the greatest idea to use **r2db** as a way of getting back to R2Db — MacsBug and R2Db aren't reentrant in all cases. Instead, the best way to get back to R2Db is by typing a simple "G" in MacsBug.

scp

The dcmd **scp** stands for "stack crawl PowerPC." Given an address, it will unwind PowerPC stack frames to display a calling sequence. This dcmd doesn't understand Mixed Mode switch frames, so it may not prove useful in the general case.

tdp

The dcmd **tdp** stands for "total dump PowerPC." Much like its MacsBug sibling **td**, **tdp** displays all the registers from the PowerPC context. This is useful for looking at a crash when the PPC Debugger Nub isn't installed.

DEBUGGING WITHOUT THE DEBUGGER NUB

Believe it or not, you can remove the PPC Debugger Nub and still debug your code. While this isn't the optimal debugging environment, you may find yourself in it in the future, and it's good to be prepared.

The basic problem is to understand the context that the unhandled native exception will put you in, what values are a good reflection on the native execution context, and how to reconstruct what happened in your mind.

Remove the debugger nub from your System Folder and reboot. Then launch CrashOMatic and choose the Bus Error command from the Debug menu. This will drop you into MacsBug with a spurious interrupt. MacsBug has no idea what a PowerPC exception frame looks like and makes this not-so-great guess as to the cause of your problem.

Start the attack by looking at the native context. (You may need the *PowerPC 601 RISC Microprocessor User's Manual* handy if you haven't done much of this yet.) Type "tdp" to get a dump that will show you the context of the native machine.

```
Spurious Interrupt or Uninitialized Interrupt Vector at 00643762
while fetching instructions from FFFFFFFE and 00000000 while reading
word from 20104802 in F

tdp
  PowerPC registers from context block
   PC = 08c877e0     LR = 08c87cf0     CR = 24000004
  CTR = 409EEE18    XER = 20000010
   r0 = 000225F8     r8 = 08CA4908    r16 = 00000000    r24 = 00000000
   r1 = 08D93DDA     r9 = 00000000    r17 = 00000000    r25 = 409DF150
   r2 = 00022904    r10 = 0003AB30    r18 = 46810000    r26 = 00133002
   r3 = deadbeef    r11 = 409EEE18    r19 = 00000000    r27 = 00000002
   r4 = deadbeef    r12 = 6802D764    r20 = 00000000    r28 = 08D93DDA
   r5 = 68FFF740    r13 = 68FFF400    r21 = 00000000    r29 = 0008C838
   r6 = 00000000    r14 = 00000000    r22 = 48400000    r30 = 0008C82C
   r7 = 08CA4808    r15 = 00000000    r23 = 00000000    r31 = 68FFF740
```

The PowerPC runtime environment specifies that R1 is used as the stack pointer and that the return address is kept inside the link register, LR. If we **dis** at the PC, we can see that we're inside the routine doWithoutProtection.

```
dis 8c877e0
  doWithoutProtection+34
   +0034  08c877e0  lwz    r5,0x0(r4)                            | 80a40000
   +0038  08c877e4  stw    r5,0x3c(SP)                           | 90a1003c
   +003c  08c877e8  b      doWithoutProtection+58                | 4800001c
   +0040  08c877ec  lha    r6,0x5a(SP)                           | a8c1005a
   +0044  08c877f0  cmpi   0,r6,0x0005                           | 2c060005
   +0048  08c877f4  bc     BO_IF,CR0_EQ,doWithoutProtection+20   | 4182ffd8
   +004c  08c877f8  lha    r7,0x5a(SP)                           | a8e1005a
   +0050  08c877fc  cmpi   1,r7,0x0004                           | 2c870004
   +0054  08c87800  bc     BO_IF,CR1_VX,doWithoutProtection+30   | 4186ffdc
   +0058  08c87804  lwz    r0,0x48(SP)                           | 80010048
   +005c  08c87808  addic  SP,SP,0x0040                          | 30210040
   +0060  08c8780c  mts    LR,r0                                 | 7c0803a6
   +0064  08c87810  bclr   BO_ALWAYS,CR0                         | 4e800020
```

Of course, it's pretty easy to debug a rigged example, but nonetheless it's a valuable experience to get practice reading hex dumps of PowerPC stack frames inside MacsBug. This actually has some historical significance, in that many folks (in the original generation of programmers) debugged problems almost exclusively through large printed stacks of octal numbers known affectionately as “dumps.” Now a new generation of hackers can learn how difficult life was when their bosses not only programmed with toggle switches but also had to walk to school through a snowstorm, uphill — both ways.

NOW BUG OFF!

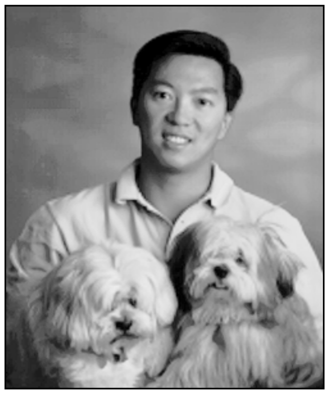
For more interesting examples of hybrid debugging, check out the Debug menu in CrashOMatic. Included at no extra charge are examples of using the PowerPC Exception Manager to do strange and exciting things with your program. We hope that these hints and techniques will help you debug your PowerPC application in less time than it took us to write this article.

REFERENCES

- “Making the Leap to PowerPC” by Dave Radcliffe, *develop* Issue 16.
- “Macintosh Debugging: The Belly of the Beast Revisited” by Fred Huxham and Greg Marriott, *develop* Issue 13.
- “Macintosh Debugging: A Weird Journey Into the Belly of the Beast” by Bo3b Johnson and Fred Huxham, *develop* Issue 8.
- Macintosh Technical Note “Memory Manager Compatibility” (Memory 13).
- *Inside Macintosh: PowerPC System Software* (Addison-Wesley, 1994).
- *PowerPC 601 RISC Microprocessor User’s Manual* (Motorola, 1993).

THANKS TO OUR TECHNICAL REVIEWERS
Jeff Cobb, Bruce Jones, Bill Kincaid, Dave Radcliffe•

Many thanks to the folks on the PowerPC development team who came up with the techniques and dcmds mentioned here. Even more thanks to the Macintosh debugging demigods who gave us things like EvenBetterBusError, Programmer’s Key, XapHandles, and MacsBug.•



JOHN WANG

SOMEWHERE IN QUICKTIME

CROSS-PLATFORM COMPATIBILITY AND MULTIPLE-MOVIE FILES

The success of QuickTime since its introduction in December of 1991 has been extraordinary; not only are there now countless applications and multimedia products for QuickTime on the Macintosh platform, but QuickTime has been successful for cross-platform developers as well. QuickTime for Windows, shipping since November of 1992, is becoming an important tool for Windows multimedia developers. Because movies can be created with QuickTime on the Macintosh and then used unaltered with QuickTime for Windows, developers are able to create a movie once and use it on both systems. This capability has been available since the 1.0 release of QuickTime.

To satisfy your curiosity and help you out with your own QuickTime products, this column discusses how to create movie files that are cross-platform compatible and then gives techniques for storing and accessing multiple movies in a single file (which currently poses a problem in the cross-platform environment).

CREATING CROSS PLATFORM–COMPATIBLE MOVIE FILES

The structure you choose for your QuickTime movie files will depend in part on the platform your movies will run on. This is because the Macintosh file system is different from other file systems in that there can be up to two forks in every file: a data fork and a resource fork. The data fork is the standard file for I/O that most file systems support. The resource fork, however, is unique to the Macintosh. It's a "database" of resources that can be randomly accessed via the Resource Manager.

There are two components to a movie file: the *movie resource atom* — a small data structure describing the movie, its tracks, and the tracks' media — and the *movie data atom*, containing the movie's data such as video and sound. A typical movie file on the Macintosh consists of a movie resource atom in the resource fork and a movie data atom in the data fork. Storing the movie resource atom in the resource fork with resource type 'moov' allows easy random access to the data it contains. The movie data atom is stored in the data fork as sequential data, which optimizes playback performance.

Because file systems on other platforms don't support a resource fork, a movie that is to be used on other platforms must have its movie resource atom stored in the data fork of the movie file. This type of file is called a *single-fork file* because the movie resource atom and the movie data atom are both stored in the data fork of the movie file, and the resource fork is not used.

Movies that will be played on both Macintosh and non-Macintosh platforms can use a file in which the movie resource atom is redundantly stored in both the resource fork and the data fork of the file. The movie data atom is, as always, stored only in the data fork. For lack of a better term, this type of file is also often called a single-fork file since the resource fork of the movie file can be ignored.

The simplest way to create a single-fork movie file for cross-platform compatibility is to use the Movie Converter application from the QuickTime Starter Kit (available from Apple-authorized dealers). The Save As dialog in this application has a checkbox that allows a movie to be stored in the single-fork format. However, the QuickTime API can easily be used to add this capability to any application. You simply call either FlattenMovie or FlattenMovieData with the flattenAddMovieToDataFork flag set. When called with this flag set, FlattenMovie places the movie

JOHN WANG (AppleLink WANG.JY) is often seen sneaking around the Apple campus with a heavy duffle bag that one would think to be dozens of 8•24 GC cards that John is trying to get out of harm's way. But, in truth, John is protecting unknowing strangers from his two attack dogs, shown above.•

resource atom in both forks, while FlattenMovieData puts it only in the data fork, leaving the resource fork untouched.

Movie files that are created with FlattenMovie or FlattenMovieData with the flattenAddMovieToDataFork flag set are compatible with both QuickTime and QuickTime for Windows because QuickTime is able to retrieve the movie resource atom and the movie data from the data fork alone on either platform.

MULTIPLE-MOVIE FILES FOR THE MACINTOSH

We'll return to cross-platform issues in a moment, but first let's take a look at the simple case of multiple-movie files: those that will be used only on the Macintosh platform. You can create such files in one of two ways, depending on whether you want the resulting file to be self-contained (that is, to contain all of the movie information in one file). Both of these methods have their benefits and drawbacks. The one that will work best for you depends on your implementation and performance testing.

To create a self-contained multiple-movie file for the Macintosh, you can use the FlattenMovie call (without the flattenAddMovieToDataFork flag set) to append an entire movie to an existing movie file. This adds the movie resource atom to the resource fork and the movie data atom to the data fork of the file.

The second method creates a multiple-movie file that's not self-contained, but instead consists entirely of movie resource atoms that reference other movie files containing the movie data atoms. This technique can be useful if, for example, you're writing an interactive CD application that uses multiple movies and you want to store all the movie resource atoms in a single file that's relatively small. This file can be copied locally to a hard disk for faster access. The drawback of this approach is that you have to handle multiple files rather than have everything in one file. The excerpt below from the sample code provided on the CD demonstrates — without error checking, for better readability — how easy it is to add an existing movie's resource atom to a movie file.

```
// Open source movie file to add.
OpenMovieFile(&reply.sfFile, &sourceRefNum,
   fsRdWrPerm);
NewMovieFromFile(&theMovie, sourceRefNum,
   &resId, movieName, 0, (Boolean *) 0);
CloseMovieFile(sourceRefNum);

// Write out to the collection file.
currentResFile = CurResFile();
UseResFile(collectionRefNum);
myResId = 0;
AddMovieResource(theMovie, collectionRefNum,
   &myResId, movieName);
UseResFile(currentResFile);

// Clean up.
DisposeMovie(theMovie);
```

AddMovieResource does more than just add the movie resource atom: it updates the data references so that the new movie will refer to the correct movie data file.

Accessing multiple-movie files for Macintosh-only use is easy. You just use the Resource Manager to access the movie resource atoms in the resource fork of the file. The following excerpt from the sample code on the CD uses the Resource Manager to access the various movies. (The code on the CD adds the resource names of all the movies to a menu; your code can of course do whatever it wants with the movies.)

```
currentResFile = CurResFile();
UseResFile(theRefNum);
movieCount = Count1Resources('moov');
for (i=1; i<=movieCount; i++) {
   movieResource = Get1IndResource('moov', i);
   // Do whatever you want with this movie.
   . . .
   ReleaseResource(movieResource);
}
UseResFile(currentResFile);
```

CROSS-PLATFORM MULTIPLE-MOVIE FILES

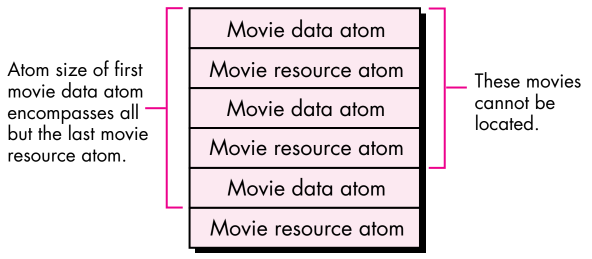
To create a multiple-movie file that's cross-platform compatible, you just repeatedly call FlattenMovie or

For information on movie resource formats, see Chapter 4 of *Inside Macintosh: QuickTime.*•

FlattenMovieData with the flattenAddMovieToDataFork flag set, as described earlier under "Creating Cross Platform–Compatible Movie Files." Accessing such a file, however, is complicated. Since it's a single-fork file, you can't access the movie resource atoms randomly as with the Resource Manager. There's also a problem with accessing the atoms directly; to understand this problem, you need to know how a single-fork movie file is structured.

The movie data atom appears before the movie resource atom in a single-fork file. The first four bytes of the movie data atom contain the size of the atom. (Figure 4-2 in *Inside Macintosh: QuickTime*, page 4-5, illustrates this structure.) This permits QuickTime to skip over the movie data atom to find the movie resource atom. If you place several movies in the same file, the movie data atoms are interleaved with the movie resource atoms.

Here's the problem: A bug in QuickTime through its current version (1.6.1) causes the size field in the movie data atoms to be filled in incorrectly. The illustration below shows how QuickTime currently creates a single-fork movie file in which FlattenMovieData has been called multiple times. The atom size in the first movie data atom takes you to the last movie resource atom rather than the first movie resource atom. Therefore, you can locate only the last movie in the file.


Atom size of first movie data atom encompasses all but the last movie resource atom.
Movie data atom
Movie resource atom
Movie data atom
Movie resource atom
Movie data atom
Movie resource atom
These movies cannot be located.

This bug will be fixed in the next version of QuickTime. The solution for now is to use the wrapper function called BetterFlattenMovieData, included on this issue's CD. This function, when called multiple times, creates a single-fork file in which the size and type fields of all atoms contain the correct information.

To locate the movies in a file with correct size and type fields, you traverse the data fork and read in those fields of each atom to determine its size and whether it's a movie resource atom. When you know the size of the atom, it's easy to skip over it and access the next atom. Of course, the last atom is reached when the end of the file is reached. The following code from the CD demonstrates how you would do this.

```
// Locate the file offset of the movie resource
// atom stored at the specified index.
if (FSpOpenDF(theFile, fsRdPerm, &myRefNum)
      == noErr) {
   currentLoc = 0;
   doneCounter = index;
   *fileOffset = 0;
   while ((doneCounter > 0)) {
      if (err = SetFPos(myRefNum, fsFromStart,
            currentLoc))
         return(err);
      readLength = 8;
      if (err = FSRead(myRefNum, &readLength,
            header))
         return(err);
      if (header[1] == 'moov')
         doneCounter--;
      *fileOffset = currentLoc;
      currentLoc += header[0];
   }
   FSClose(myRefNum);
} else
   return (err);
return (noErr);
```

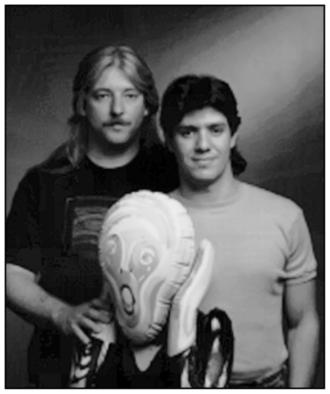
MOVING ON

I'm happy to see that QuickTime is being used in new and exciting ways. By taking full advantage of the cross-platform and multiple-movie capabilities of QuickTime, you can bring your products to new heights. The sample code on the CD will start you on your way. Enjoy!

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

CONCURRENT PROGRAMMING WITH THE THREAD MANAGER

Let us introduce you to the latest addition to the Macintosh Toolbox — the Thread Manager. The Thread Manager enables concurrent programming so that you can provide true multitasking within your application. We give a quick overview of the Thread Manager and then move on to discuss advanced programming techniques not found in the Thread Manager documentation.



ERIC ANDERSON AND BRAD POST

The Thread Manager is a new part of the Macintosh Toolbox that provides both cooperative and preemptive threads of execution within an application. Although it's available only within your application and is not used for systemwide preemption, you can take advantage of it in many valuable ways:

- Use a threaded About box so that your application can continue running in the background while displaying a modal dialog.
- Do anything you do today at idle time with null events within a thread. This avoids the complexity of writing idleProcs.
- Decouple time-consuming processes from the user interface of your application. For example, create an image-rendering thread for each image to render, and use the main thread for the user interface of the application. Similarly, with graphics screen redraws and print spooling, have a thread do the redraw or spool a print job while the main thread handles the user interface.
- Construct complex simulations without complex logic. For example, in a program that simulates city streets, use a thread for each car, one for each traffic signal, and one for the time of day.
- Use threaded I/O and communication to easily allow an application to act as both a client and a server. With this approach, applications can handle incoming questions and wait for incoming answers simultaneously. Examples of this are discussed in more detail in "Threads on the Macintosh" in *develop* Issue 6.

ERIC ANDERSON has been developing various parts of the Macintosh Toolbox and OS for over five years. Having worked on System 7 virtual memory for the past four years and the Thread Manager for the past two, he thinks that moving to Hawaii and building boats is an excellent idea. Before Apple, he designed tower and street crane-positioning simulation software and developed various software/hardware systems for controlling live-action effects (read: pyrotechnics) for film productions.•

The Thread Manager has all the rights and privileges of other services in the Macintosh Toolbox, such as a trap interface to avoid linking a library into your code and header files for C, Pascal, and assembly-language programmers. It's a fully supported product and will be around for years to come. You can license the Thread Manager through Apple's Software Licensing group (AppleLink SW.LICENSE). The Thread Manager works on all Macintosh platforms running System 7 or later.

THREAD MANAGER OVERVIEW

This section describes the two types of threads — cooperative and preemptive — and the basic services for creating, scheduling, and deleting threads and gathering thread status. It also discusses the main thread, code organization, and thread disposal.

COOPERATIVE THREADS

Cooperative threads allow cooperative multitasking. They're the easiest to use in terms of scheduling and accessibility to Toolbox traps. Everything you can do today in a standard application you can do with a cooperative thread — memory allocation, file I/O, QuickDraw, and so on. Cooperative threads yield to other cooperative threads only when the developer explicitly makes one of the Thread Manager yield calls or changes the state of the current cooperative thread.

Cooperative threading in the Thread Manager is similar to the cooperative threading in the Threads Package made available through APDA a few years ago (see "Threads on the Macintosh" in *develop* Issue 6). In fact, this library should no longer be used since the Thread Manager is replacing it as the preferred method. Converting your applications from the Threads Package to the Thread Manager is easy as long as you don't rely heavily on the internal data structures provided by the Threads Package. The big advantage to using the Thread Manager is that thread stacks are register swapped, not block moved, during a context switch.

PREEMPTIVE THREADS

Preemptive threads allow true multitasking at the application level. Whenever the application gets time from the Process Manager, preemptive threads for that application are allowed to run. Unlike cooperative threads, which execute only when a previously running cooperative thread explicitly allows it, preemptive threads may interrupt the currently executing thread at any time to resume execution. You can make the preemptive thread yield back to the just-preempted cooperative thread with any of the Thread Manager yield calls. Alternatively, a preemptive thread can simply wait for its time quantum to expire and automatically yield back to the cooperative thread it interrupted. If the interrupted cooperative thread is in the stopped state, the next available preemptive thread is made to run. Preemptive threads then preempt each other, in a round-robin fashion, until the interrupted cooperative thread is made ready. Figure 1 illustrates the default round-robin scheduling mechanism for threads.

BRAD POST (AppleLink BPOST) is a member of SMMFD (a geek club for mathematical modeling and fractal design). His other hobbies include computer graphics, skiing, surfing, playing Ultimate Frisbee, drinking Tied House Ale, working out, and just living life to its fullest. Remember, no one gets outta here alive!•

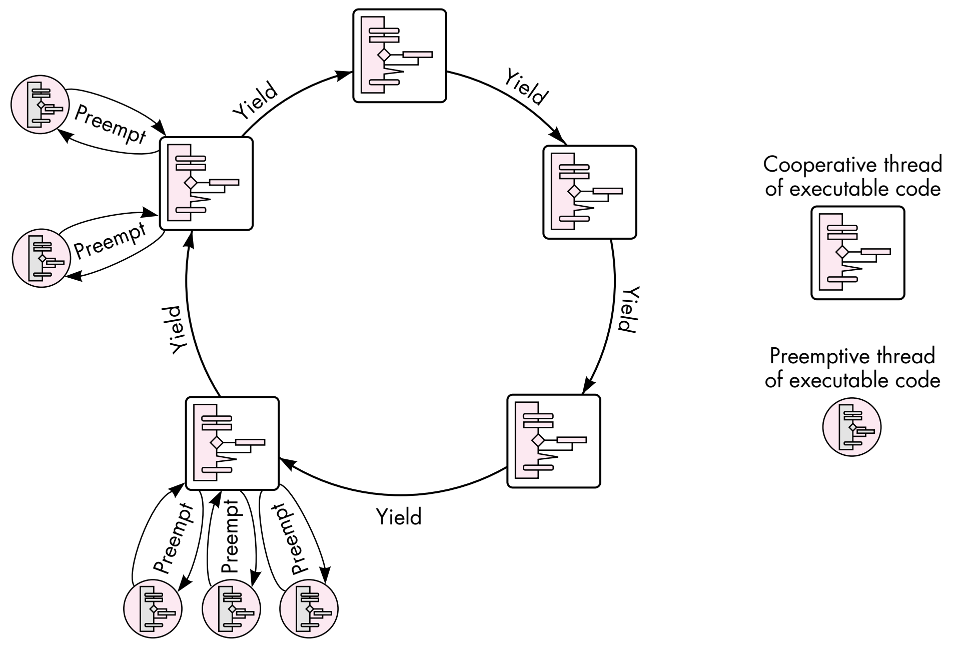

Yield
Yield
Yield
Yield
Yield
Yield
Preempt
Preempt
Preempt
Preempt
Preempt
Cooperative thread of executable code
Preemptive thread of executable code

Figure 1
Round-Robin Scheduling Mechanism

For situations where you don't want a thread to be preempted, the Thread Manager provides two calls for turning off preemption (see the next section). These calls don't disable interrupts, just thread preemption.

Because a preemptive thread can usually "interrupt" cooperative threads and doesn't need to explicitly use the API to yield to other threads, preemptive threads have to conform to the guidelines set up for code executed at interrupt time. Don't make any calls that are listed in *Inside Macintosh X-Ref*, Revised Edition, Appendix A, "Routines That Should Not Be Called From Within an Interrupt." No moving of memory is allowed and definitely no QuickDraw calls should be made. QuickDraw calls may seem tempting and may even appear to work, but they fail on many occasions and can corrupt other parts of memory in subtle ways that are very difficult to debug. (QuickDraw GX was designed to be reentrant, so preemptive threads may make use of the QuickDraw GX feature set if they follow the rules set up by QuickDraw GX.) If there's only one thing you learn in this article, make sure it's this: *preemptive threads must follow interrupt-time rules!*

THE THREAD MANAGER API

There are several data types declared in the Thread Manager API that determine the characteristics of a thread. These include the type of thread you're dealing with (cooperative or preemptive), the state of a thread (running, ready, or stopped), and a set of options for creating new threads. With the thread creation options, you can create a thread in the stopped state (to prevent threads from beginning execution before you want them to), create a thread from a preallocated pool of threads (which is how you would create a new thread from a preemptive thread), or tell the scheduler not to save FPU state for the new thread (to reduce scheduling times for threads that don't use the FPU). These creation options are combined into one parameter and passed to the routines that create new threads.

General-purpose services make it possible to create a pool of preallocated threads and determine the number of free threads in the pool, get information on default thread stack sizes and the stack currently used by a thread, and create and delete threads.

There are basic scheduling routines for determining the currently running thread and for yielding to other threads. You can also use the preemptive scheduling routines — ThreadBeginCritical and ThreadEndCritical — to define critical sections of code. A critical section of code is a piece of code that should not be interrupted by preemptive threads — because it's changing shared data, for example. The use of critical sections of code is needed to prevent interference from other threads and to ensure data coherency within your code. Note that preemptive threads run at the same hardware interrupt level as a normal application, and *calls to ThreadBeginCritical don't disable hardware interrupts; they simply disable thread preemption.*

Advanced scheduling routines enable you to yield to specific threads and to get or set the state of nearly any thread. (You can't set a thread to the running state or change the state of the currently running thread if it's in a critical section of code.) Custom context-switching routines allow you to add to the default thread context and may be associated with any thread on a switch-in and/or switch-out basis. Any thread may have a custom context switcher for when it gets switched in and a different switcher for when it gets switched out. In addition, a custom scheduling procedure may be defined that gets called every time thread rescheduling occurs. All of these features may be used by both types of threads.

The Thread Manager also provides debugging support: a program or debugger can register callback routines with the Thread Manager so that it gets notified when a thread is created, deleted, or rescheduled.

THE MAIN THREAD

When an application launches with the Thread Manager installed, the Thread Manager automatically defines the application entry point as the main cooperative thread. The main cooperative thread is commonly referred to as the *application thread* or *main thread*. There are several guidelines you should follow regarding the main thread. These aren't hard and fast rules, but just guidelines to keep you from having debugging nightmares. Remember that cooperative threads must make explicit use of the Thread Manager API to give time (yield) to other cooperative threads.

First, the Thread Manager assumes the main thread is the thread that handles the event mechanism (WaitNextEvent, GetNextEvent, ModalDialog, and so on). Events that can cause your application to quit should be handled by the main thread: the application was entered through the main thread and should exit through it as well. The Thread Manager schedules the main thread whenever a generic yield call is made and an OS event (such as a key press or mouse click) is pending. This is done to speed up user-event handling; otherwise, it could take a long time before the main thread got time to run, because of the round-robin scheduling mechanism used for cooperative threads. In general, *all* event handling should be done from the main thread to prevent event-handling confusion. For example, if the main thread calls WaitNextEvent while another thread is calling GetNextEvent while yet another thread is calling ModalDialog, and they're all yielding to each other, sorting out which event belongs to which thread becomes a nightmare. Just let the main thread handle all events. Your application will run faster and work better in the long run.

The second guideline is to avoid putting the main thread in a stopped state. Doing so is perfectly legal but can lead to some exciting debugging if your application makes incorrect assumptions about the state of the main thread.

Last but not least, be sure you call MaxApplZone from the main thread to fully extend your application heap before any other cooperative threads get a chance to allocate, or move, memory. This requirement is due to the limitation on cooperative threads extending the application heap — they can't.

DISPOSING OF THREADS

When threads finish executing their code, the Thread Manager calls DisposeThread. Preemptive threads are recycled to avoid moving memory. Recycling a thread causes the Thread Manager to clear out the internal data structure, reset it, and place the thread in the proper thread pool. (There are two pools, one for cooperative and one for preemptive threads.) Since recycling a thread doesn't move memory, preemptive threads can dispose of themselves and other threads safely.

The Thread Manager handles disposing of threads automatically, so it's not necessary for the last line of your code to call DisposeThread unless you want to — for example, to recycle all terminating cooperative threads. DisposeThread takes three parameters: the thread ID of the thread to be disposed of, a return value to return to the creator of the thread, and a Boolean flag that tells the Thread Manager whether or not to recycle the thread. A preemptive thread should only call DisposeThread with the recycle flag set to true.

The Thread Manager API also defines a routine, SetThreadTerminator, that allows your application to install a callback routine to be called when a thread terminates — which is when it's disposed of. You could use SetThreadTerminator to do any cleanup needed by the application, but remember, you can't move memory during termination of a preemptive thread because you're still in the preemptive state.

When your application terminates, threads are terminated as follows:

- Stopped and ready threads are terminated first but in no special order.
- The running thread, which should be the main thread, is always terminated last.

CODE ORGANIZATION

To use the Thread Manager most effectively, break your applications into discrete functional pieces, similar to the code organization used for object-oriented applications. Keep the code for event handling, window management, calculation, and drawing separate. In general, design your code such that the discrete parts may run independently of each other, but in an organized and coherent way. Object-oriented code, in general, is designed this way and is much easier to think about in a concurrent manner. The following sections discuss techniques that you can use to write more effective thread code. The examples illustrate what we learned from lots of trial and error during the development of the Thread Manager, which should reduce the development time for your threaded applications.

THREAD INPUT AND OUTPUT PARAMETERS

When creating a thread, you can pass in both a thread input parameter and a return value storage location. The Thread Manager defines the input parameter as a void* and the return parameter as a void**. Don't let the messy C-language semantics get you down; it's actually quite easy to understand.

The Thread Manager assumes the input parameter is the size of a void* (a 32-bit long word for today's 680x0 applications) and is passed to your thread as such. Any value may be passed as the input parameter, as long as it's the size of a void*. Passing nil as this parameter will simply pass nil — not nothing — as your thread's input parameter.

The return parameter is a little messier. In asking for a void**, the Thread Manager is asking for the memory location to store the void* returned by the thread. If nil is passed as the return value storage location, the Thread Manager assumes there's no return value and won't bother to set it on thread completion. Your thread may return any value as long as it's the *size* of a void*. The address passed in as the return value storage location can be simply the address of a long-word local variable (a long word is the size of a void*) or can be as complicated as the address of a global variable record pointer. The idea is that the Thread Manager wants, as the return value storage location, the address in which to store a value the size of a void*. The Thread Manager stores the returned value at thread termination time by using either a return (x) statement or a DisposeThread call.

Don't pass the address of a short-word local variable as the return value storage location, because a short word is shorter than a void*. When the thread terminates,

The return value has been set by the time a thread termination routine is executed. In this way, a piece of code may be notified when the return result of a terminated thread becomes valid. There's more information on thread termination routines in the Thread Manager documentation.•

the Thread Manager will slam all 32 bits of the return value (the size of a void*) into the short word, destroying the high word of your next defined variable. You must also make sure the storage location used for the return parameter is valid at the time the thread terminates. For example, if you use a local variable as the storage location, you need to be sure the new thread will terminate before you return from the routine that created it (that is, before the local variable goes out of scope).

The following code samples show two methods of sending in thread parameters and returning thread results. First let's define some constants and the record structure used in the example:

```
#define  kNoCreationOptions         0  /* Just use the standard default */
                                       /* creation options.             */
#define  kDefaultThreadStackSize    0  /* Zero tells the Thread Manager */
                                       /* to use the default value.     */

struct ExampleRecord {
   long     someLongValue;
   short    someShortValue;
   };
typedef struct ExampleRecord ExampleRecord;
typedef ExampleRecord *ExampleRecordPtr;
```

The kNoCreationOptions #define specifies the default creation options. These options create a thread using newly allocated memory from the Memory Manager with the thread in the "ready" state; the thread will have its FPU context switched during a reschedule. The kDefaultThreadStackSize #define tells the Thread Manager to use the default stack size for the thread.

Now let's look at the routine prototypes and thread parameters:

```
pascal long ExampleOne (long);
pascal ExampleRecordPtr ExampleTwo (ExampleRecordPtr inputRecordParam);

void ParametersExample (void)
{
   ThreadID            tempThreadID;
   OSErr               err;
   long                myLong;
   short               myShort;
   Boolean             notDone = true;
   long                longResult = -1;
   ExampleRecordPtr    recordOutResult = nil;
   ExampleRecord       recordInParam;
```

In the call to create the first thread, the numeric value 42 is passed in as the input parameter. The return value storage location is the address of a long-word local variable. Remember, you can't return from this function until the new thread terminates — the local variable return value storage must stay valid.

```
err = NewThread(kCooperativeThread, (ThreadEntryProcPtr)(ExampleOne),
   (void*)42, kDefaultThreadStackSize, kNoCreationOptions,
   (void**)&longResult, &tempThreadID);
if (err)
   DebugStr("\p Could not make coop thread 1");
```

To create the second thread, the address of a locally allocated data structure is passed as the input parameter. The return value storage location is the address of a local variable used to hold a pointer to the type of data structure to be used for storing the return value. Again, you can't return from this function until the new thread terminates.

```
recordInParam.someLongValue = 0xDEADBEEF;
recordInParam.someShortValue = 0xBADD;
err = NewThread(kCooperativeThread, (ThreadEntryProcPtr)(ExampleTwo),
   (void*)&recordInParam, kDefaultThreadStackSize, kNoCreationOptions,
   (void**)&recordOutResult, &tempThreadID);
if (err)
   DebugStr("\p Could not make coop thread 2");
```

Now that the two threads are created and ready to run, the code sits in a loop handling user events, yielding, and waiting for the threads to complete their work.

```
while (notDone)
   {
   YieldToAnyThread();        /* Other threads run. */
```

This code looks to see whether the first thread has completed and, if it has, goes off and does what computers do best — compute on the data.

```
      if (longResult != -1)
         {
         GoHandleThreadReturnResult(longResult);
         longResult = -1;              /* Reset the value. */
         }
```

Now the code checks to see whether the second thread has completed. If there's a non-nil value in recordOutResult, the data is available. It's then sucked out of the record into the local variables and sent on its way to be used by the application. Notice that the record structure that was allocated by the second thread is disposed of.

```
      if (recordOutResult != nil)
         {
         myLong = recordOutResult->someLongValue;
         myShort = recordOutResult->someShortValue;
         DoStuffWithParams(myLong, myShort);
         DisposePtr((Ptr)recordOutResult);      /* Toss it. */
         recordOutResult = nil;                 /* Neutralize it. */
         }
      /* Handle user events until we quit. */
      GoHandleEvents(&notDone);        /* WaitNextEvent event handling. */
      }
   return;                             /* Out of here. */
}
```

In the code for the first thread, both the input and output parameters are longs, and the thread simply returns the value sent to it. The important point here is that both the input and output parameters are the size of a void*.

```
pascal long ExampleOne (long inputParam)
{
   return (inputParam);    /* Must be the size of a void*. */
}
```

The code for the second thread is a little more complex. Here, we allocate the storage needed for the record that will be passed back to the creator. It's the creator's duty to dispose of the data after using it. The record elements from the input parameter are used as the source material for the elements within the record being returned to the creator. Again, both the input and output parameters are the size of a void*.

```
pascal ExampleRecordPtr ExampleTwo (ExampleRecordPtr inputRecordParam)
{
   ExampleRecordPtr     myRecordPtr;

   myRecordPtr = NewPtr(sizeof(ExampleRecord));
   myRecordPtr->someLongValue = inputRecordParam->someLongValue;
   myRecordPtr->someShortValue = inputRecordParam->someShortValue;

   return (myRecordPtr);   /* Must be the size of a void*. */
}
```

PARANOID PREEMPTIVE PROGRAMMING

If a preemptive thread can preempt, it will preempt. This can cause problems, especially when you create new preemptive threads. When you create a preemptive thread in the ready state, it's free to run whenever preemption occurs. This could mean that your preemptive thread fires off before data it might need is available or set up. There are two simple solutions, illustrated in the examples that follow: create the preemptive thread in the stopped state and wake it up when you're ready to let it run, as shown in the first example, or wrap all creation of "ready" preemptive threads within critical sections, as shown in the second example. Remember that preemptive threads can't preempt code that's wrapped within critical sections.

Here's the example of creating a preemptive thread in the stopped state:

```
pascal void* aThread (void* parameter)
{
   /* Preemptive threads don't need to call Yield. */
   while (true);     /* Infinite loop. */
}

void somefunction (void)
{
   ThreadID theThreadID;
   OSErr    theError;

   theError = NewThread(kPreemptiveThread,        /* Type of thread */
                        aThread,                  /* Thread entry point */
                        nil,                      /* Input parameter */
                        kDefaultThreadStackSize,  /* Stack size */
                        kNewSuspend,              /* Creation options */
                        nil,                      /* Return result */
                        &theThreadID);            /* New thread ID */
   if (theError != noErr)
      DebugStr("\pFailed to create thread");
   . . .
   /* Now wake the thread up. */
   theError = SetThreadState(theThreadID,         /* Thread to set */
                             kReadyThreadState,   /* New state */
                             kNoThreadID);        /* Suggested thread */
   if (theError) != noErr
      DebugStr("\pFailed to set the preemptive thread to the ready
         state.");
}
```

The parameter kNewSuspend in the NewThread call tells the Thread Manager to create the thread in the stopped state. The SetThreadState call is used to wake up the thread. The last parameter to SetThreadState suggests to the Thread Manager which thread to execute after this call; this parameter is used only if the state of the currently executing thread is being changed. At this point, the preemptive thread is allowed to begin execution whenever it can.

Here's how to create a preemptive thread within a critical section:

```
pascal void* aThread (void *parameter)
{
   /* Preemptive threads don't need to call Yield. */
   while (true);     /* Infinite loop. */
}

void somefunction (void)
{
   ThreadID theThreadID;

   ThreadBeginCritical();  /* Disable preemption. */

   theError = NewThread(kPreemptiveThread, aThread, nil,
      kDefaultThreadStackSize, kNoCreationOptions, nil, &theThreadID);
   if (theError != noErr)
      DebugStr("\pFailed to create a preemptive thread");

   ThreadEndCritical();    /* Enable preemption. */
}
```

AVOIDING QUICKDRAW FROM PREEMPTIVE THREADS

As mentioned earlier, you can't call QuickDraw from preemptive threads because QuickDraw isn't interrupt safe and you have to obey the rules of executing during interrupt time when you use preemptive threads. There's no way around this, but you *can* draw from preemptive threads with some extra work.

First you need to write your own primitives for drawing. This can be as simple as blasting pixels with *(pixelPtr) = 0xFFFF or as complex as writing your own primitives for all drawing. Next you create a GWorld for your preemptive thread to draw into. The idea is to modify this GWorld and then later use CopyBits from a cooperative thread to blast it to the screen. There are just a couple of things to remember before you start using the preemptive thread's GWorld: make sure you lock down the PixMap handle within the GWorld using HLock and make sure you call LockPixels for that GWorld. This guarantees there won't be any memory moving while you're drawing into the GWorld with preemptive threads.

APPLICATION/THREAD MANAGER SCHEDULING

This section provides code examples for creating your own scheduler and context-switching routines. Figure 2 shows how your custom scheduler and custom context-switching routines are executed in conjunction with the Thread Manager's own scheduling routines.

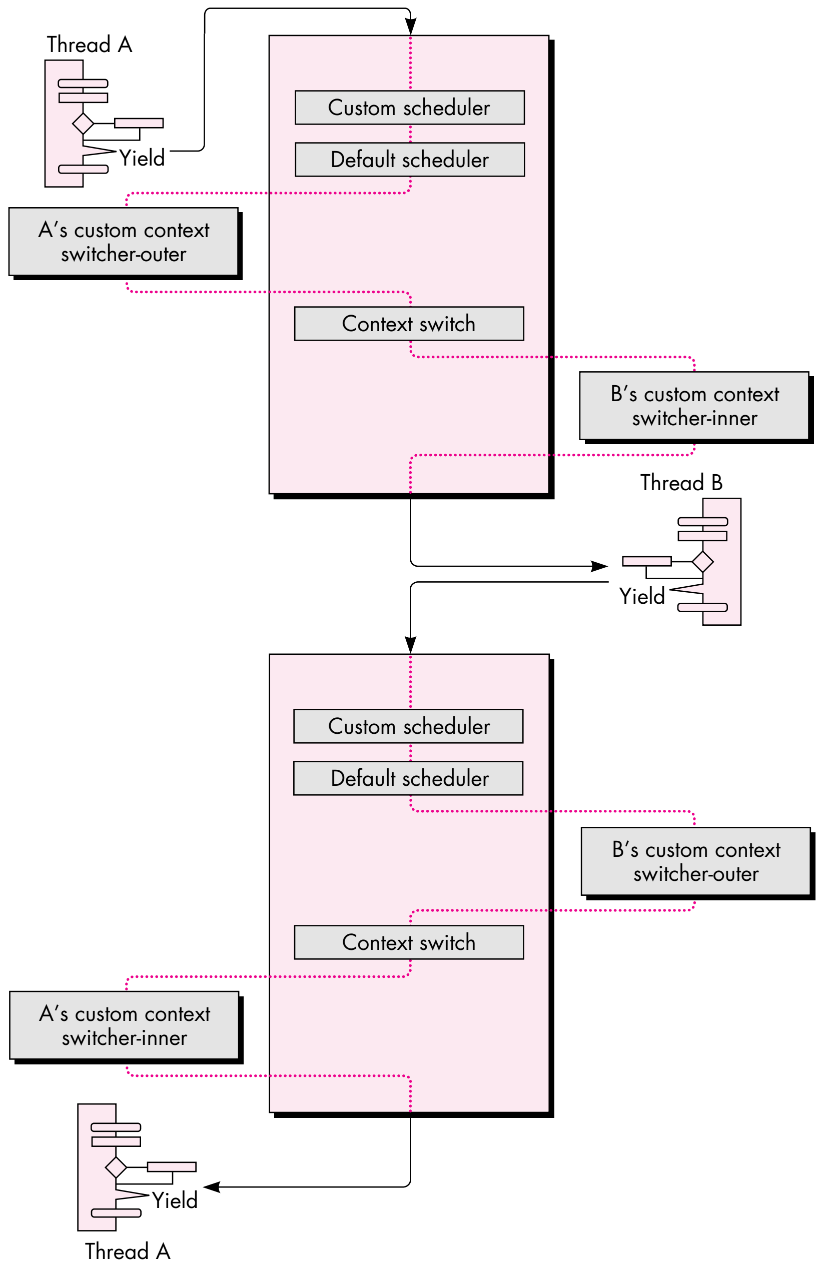


Figure 2
Thread Scheduling With Custom Scheduler and Context Switcher

CUSTOM SCHEDULERS

Occasionally, the YieldToThread(ThreadID) call is insufficient as a thread-scheduling mechanism. Assigning a custom scheduler function gives localized control over thread scheduling within your application. From this function you can inform the Thread Manager which thread you would like to schedule. Here's the header information that deals with custom schedulers (it's extracted from Threads.h):

```
/* Information supplied to the custom scheduler */
struct SchedulerInfoRec {
   unsigned long  InfoRecSize;
   ThreadID       CurrentThreadID;
   ThreadID       SuggestedThreadID;
   ThreadID       InterruptedCoopThreadID;
};
typedef struct SchedulerInfoRec SchedulerInfoRec;
typedef SchedulerInfoRec *SchedulerInfoRecPtr;

typedef pascal ThreadID (*ThreadSchedulerProcPtr)(SchedulerInfoRecPtr
   schedulerInfo);
```

Note that the Thread Manager passes the custom scheduler the ID of the current thread (CurrentThreadID) and the ID of the thread that the Thread Manager is going to schedule (SuggestedThreadID). If a cooperative thread was preempted and has not yet resumed execution, the ID of that thread (InterruptedCoopThreadID) is passed to the custom scheduler; kNoThreadID is passed if there's no cooperative thread waiting to resume from being preempted. With this information, all you have to do is tell the Thread Manager which thread to execute next by returning the thread ID of the thread you want to run.

There's one rule you must follow when returning a thread ID: if the value of InterruptedCoopThreadID is not equal to 0, return the InterruptedCoopThreadID value or the ID of any ready preemptive thread. This is because scheduling a different cooperative thread while another cooperative thread has been preempted would cause cooperative thread preemption and could result in a system misunderstanding (crash). If you don't care which thread the Thread Manager schedules next, you can just return the constant kNoThreadID. If the value of CurrentThreadID is kNoThreadID, the Thread Manager schedules the first available ready thread by default; your custom scheduler may override this.

During the execution of a custom scheduler, certain conditions exist: preemption is disabled to take care of any reentrancy problems and your A5 world is not guaranteed to be valid. You're probably thinking, "Geez, I have to worry about A5!" We show you a quick way to get around that in the example below, by using Gestalt to store A5. A few more things about custom schedulers: You can't yield from within a custom scheduler, nor can you cause scheduling to recur from this function. Also, only ready and running threads are valid for rescheduling — and you can't call SetThreadState from the custom scheduler.

OK, enough background — here's the custom scheduler example:

```
#include "GestaltValue.h"  /* Gestalt utility. */

pascal ThreadID myThreadScheduler (SchedulerInfoRecPtr mySchedulerInfo)
{
   long          currentA5, myA5;
   ThreadTaskRef  currentTaskRef;

   /* The task ref is our Gestalt selector. */
   GetThreadCurrentTaskRef(&currentTaskRef);

   /* Get application's A5. If we can't, let Thread Manager do it all. */
   if (Gestalt(currentTaskRef, &myA5) != noErr)
      return kNoThreadID;

   /* Was a cooperative thread preempted? If so, let Thread Manager */
   /* continue. */
   if (mySchedulerInfo->InterruptedCoopThreadID != kNoThreadID)
      return mySchedulerInfo->InterruptedCoopThreadID;

   /* Restore application's A5. */
   currentA5 = SetA5(myA5);

   /* Now you can determine what you want to do. You have access to all */
   /* your globals, so select a thread to run. */
    . . .
   /* Restore the A5 we entered with. */
   myA5 = SetA5(currentA5);
}

void InitAll (void)
{
   long          myA5;
   ThreadTaskRef  currentTaskRef;
   OSErr         myError;

   /* Do standard initialization stuff here. */
   . . .
   myA5 = SetCurrentA5();
   /* Get a unique value to use as a gestalt selector. */
   GetThreadCurrentTaskRef(&currentTaskRef);
```

The GestaltValue library, available on this issue's CD, provides the NewGestaltValue, ReplaceGestaltValue, and DeleteGestaltValue functions.•

```
   /* Set up a Gestalt value to use. */
   if ((myError = NewGestaltValue(currentTaskRef, myA5)) != noErr)
   {
      /* Does it already exist? It better not! */
      if (myError == gestaltDupSelectorErr)
         DebugStr("\pWon't replace Gestalt selector.");
   }
}
```

CUSTOM CONTEXT SWITCHERS

Now we come to the next feature the Thread Manager supplies for dealing with scheduling: custom context switchers. You might ask, what is a thread context? The default context of a thread consists of the CPU registers, the FPU registers if an FPU is present, and a few internal globals. The saved data is as follows:

- CPU registers: D0–D7; A0–A7; SR (including CCR)
- FPU registers: FPCR, FPSR, FPIAR; FP0–FP7

The thread context lives on the thread's A7 stack, and the location of the thread context is saved at context switch time. Initially, the A5 register (which contains a pointer to the application's A5 world) and the thread MMU mode (24-bit or 32-bit) is set to be the same as the main thread. This allows all threads to share in the use of the application's A5 world, which gives threads access to open files and resource chains, for example. To allow preemption of threads that change the MMU operating mode, the MMU mode is saved as part of the context of a thread. The FPU context is fully saved along with the current FPU frame.

Custom context switchers don't need to worry about saving the standard CPU context, as this is done by the Thread Manager. When writing a custom context switcher, keep in mind that preemption is disabled while it executes. Just as in custom schedulers, don't make any calls that can cause scheduling. In addition, when the custom context switcher is being called, the thread context is in a transition state, so any calls to GetCurrentThread won't work, nor will any calls to which you pass kCurrentThreadID as a parameter. As with the custom scheduler, you're not guaranteed to have A5 set up for you; the example below shows how to get around that.

You may have context switchers for when a thread is entered (*switcher-inner*) and when a thread is exited (*switcher-outer*). Which context switcher to use is determined by a parameter passed to the SetThreadSwitcher routine. Here's the header information on custom context switchers (extracted from Threads.h):

```
typedef pascal void (*ThreadSwitchProcPtr)(ThreadID threadBeingSwitched,
   void *switchProcParam);
```

```
pascal OSErr SetThreadSwitcher (ThreadID thread, ThreadSwitchProcPtr
   threadSwitcher, void *switchProcParam, Boolean inOrOut);
```

Besides the selector for whether the switcher is an inner or an outer, the application can supply a parameter to the switch function. This parameter may vary for every thread, allowing you to write only one custom context switcher. It isn't necessary to have a switcher for both entry and exit, and all threads can share the same switcher. One last thing to remember about switchers is that a switcher-inner is called immediately before the thread begins execution at the entry point.

Here's an example of a custom context switcher:

```
#define  switcherInner     true
#define  switcherOuter     false

struct myContextSwitchParamStruct
{
   long  appsA5;     /* Save application's A5. */
   /* Anything else you may need for your custom context switch. */
   . . .
};
typedef struct myContextSwitchParamStruct myContextSwitchParamStruct;
struct myContextSwitchParamStruct gMyContextSwitchParam;

pascal void* aThread (void *parameter)
{
   /* Does something. */
   YieldToAnyThread();
   . . .
}

pascal void myThreadSwitchProc (ThreadID theThreadBeingSwitched,
   void *theSwitchProcParam)
{
   long  currentA5, myA5;  /* A5 when our custom switcher was called. */

   myA5 = ((myContextSwitchParamStruct *)theSwitchProcParam)->appsA5;
   /* Restore application's A5. */
   currentA5 = SetA5(myA5);
   /* Now you can determine what you want to do. You have access to all */
   /* your globals, so do any context stuff. */
   . . .
   /* Restore the A5 we entered with. */
   myA5 = SetA5(currentA5);
}
```

```
void InitAll (void)
{
   ThreadID theThreadID;
   OSErr    theError;

   /* Do standard initialization stuff here. */
   . . .
   /* Set up A5 in switch parameter. */
   gMyContextSwitchParam.appsA5 = SetCurrentA5();
   /* Create a thread. */
   theError = NewThread(kCooperativeThread, aThread, nil,
      kDefaultThreadStackSize, kNoCreationOptions, nil, &theThreadID);
   if (theError != noErr)
      DebugStr("\pFailed to create a cooperative thread");
   theError = SetThreadSwitcher(theThreadID, myThreadSwitchProc,
      (void*)&gMyContextSwitchParam, switchInner);
   if (theError != noErr)
      DebugStr("\pFailed to set custom context switcher-inner");
   . . .
}
```

DIALOGS THAT YIELD

Aren't you tired of having your dialogs hang up your application while you wait for the user to do something? With the Thread Manager you can alleviate this problem by making only one call, YieldToAnyThread. Basically, if your program is broken into multiple threads, you can put the dialog handler in one of those threads (usually the main thread is best) and have the dialog's filter procedure call YieldToAnyThread. This gives any other threads that are waiting time to run. When the user makes a choice in the dialog, the main thread is scheduled automatically by the Thread Manager so that your application can handle that event immediately. While the user is getting a cup of coffee, your application can finish whatever other tasks are appropriate, without slowing the machine down.

In the following example, YieldFilter is a filter procedure for a dialog that just calls YieldToAnyThread, to give other threads time while the dialog is in the front. This allows threads "behind" the dialog to continue to process information.

```
pascal boolean YieldFilter (DialogPtr theDialogPtr, EventRecord *theEvent,
   short *theItemHit)
{
   /* Yield to whoever wants to run. */
   YieldToAnyThread();
   /* Call the 7.0 standard filter procedure defined in Dialogs.h. */
   return (StdFilterProc(theDialogPtr, theEvent, theItemHit));
}
```

```
/* The DoOKDialog function just handles a simple OK dialog. */

void DoOKDialog (short dialogID)
{
   DialogPtr   theDialog;
   short       itemHit;
   GrafPtr     savePort;
   OSErr       theError;

   GetPort(&savePort);

   if ((theDialog = GetNewDialog(dialogID, NULL, (Ptr)-1)) != NULL)
   {
      SetPort(theDialog);
      ShowWindow(theDialog);
      do
      {
         ModalDialog(YieldFilter, &itemHit);
      } while (itemHit != okButton);
      DisposDialog(theDialog);
   } else
      DebugStr("\pCould not find dialog");

   SetPort(savePort);
}
```

THREAD-BLOCKING I/O

In general, thread-blocking I/O occurs when code making an I/O call sleeps until the I/O call completes. The basic problem with doing thread-blocking I/O with the Thread Manager is that threads are application based, not systemwide Device Manager I/O based. This means that threads are really good at doing work at the application level but aren't designed to auto-block on I/O — that's your job. Fortunately, the Thread Manager provides all the tools needed to create your own thread-blocking I/O.

I/O WITH COOPERATIVE THREADS

Let's suppose you use a thread to make an asynchronous I/O call. Then the thread goes to sleep, waiting for the completion routine to wake it up. The problem is the completion routine can (and will) fire off before the thread is able to be put to sleep — there's a window of misfortune after the thread makes the asynchronous call and before it completes the sleep call. Having nothing to do, the completion call just returns. The thread then makes the sleep call and will sleep forever, waiting for a completion routine that already occurred.

A completion routine, or any other code running on the Macintosh, may ask for the state of a thread in an application by using GetThreadStateGivenTaskRef. Given a task reference to a particular application and a thread ID, you can get the state of any thread. If that thread is sleeping, you can call SetThreadReadyGivenTaskRef, which will tell the Thread Manager to mark the thread as ready. The thread is not actually *made* ready, just *marked* ready. The next time a reschedule occurs in the application, the marked thread is *made* ready and is eligible to run.

Note that you can't just check to see if the thread isn't in the running state because it could have been preempted (and probably was if possible) before it made the sleep call, when the completion routine fired. You must wait for the thread to be in the stopped state before making a call to SetThreadReadyGivenTaskRef.

One solution is to have the completion routine use one of the interrupt-safe routines and ask the Thread Manager if the thread in question is sleeping yet and, if it is, ask the Thread Manager to wake it up. If it's not sleeping yet, set up a timer task (or VBL or Deferred Task Manager task) to ask again later, and keep asking until the thread is sleeping; then wake it up.

As you can imagine, the "poll until thread stopped" solution is messy and time consuming. A cleaner solution is to have a second thread whose only job in life is to wake up the first thread after it goes to sleep (after making the asynchronous I/O call). The following steps show how to do this for the two threads ThreadA and ThreadB.

From a cooperative thread — ThreadA:

1. Create a cooperative thread, ThreadB, in the stopped state.
2. Set the completion routine to wake up ThreadB when it fires off.
3. Make the asynchronous I/O call from ThreadA.
4. Put ThreadA to sleep, forcing a reschedule with the SetThreadState call.
5. After ThreadB wakes up ThreadA, ThreadA magically continues running right where it left off! This is what's fun with concurrent programming.

From the completion routine:

1. Be safe and make the GetThreadStateGivenTaskRef call on ThreadB.
2. If ThreadB is *not* in a stopped state, give up. Things are in a bad way. ThreadB should be in the stopped state, since it was created that way.

3. Make a call to SetThreadReadyGivenTaskRef on ThreadB to *mark* it ready. The Thread Manager will actually *make* it ready at the next reschedule time.
4. That's it; just return.

While executing code from the cooperative thread, ThreadB, you know the following to be true:

- ThreadB is cooperative.
- You're executing code in ThreadB.
- ThreadA is cooperative.
- Only one cooperative thread may run at a time.
- ThreadA didn't yield to anybody and is therefore not in the ready state while ThreadB is running.
- ThreadA put itself in the stopped state, which caused a reschedule.
- Since cooperative ThreadB is running, ThreadA can't be running.
- Therefore, ThreadA is in the stopped state.

From the above truths, all ThreadB needs to do is set the thread state of ThreadA to ready. The following steps wake up ThreadA from ThreadB:

1. Set the state of ThreadA to ready with the SetThreadState call.
2. Return.

The big concept is that only one cooperative thread can run at a time. Cooperative threads can't preempt each other. The only way to get from one cooperative thread to another is by yielding — making a yield call or setting the state of the running cooperative thread to ready or stopped. If a cooperative thread, ThreadA, sets itself to stopped, another cooperative thread, ThreadB, knows that because it (ThreadB) is running, ThreadA must be stopped. However, this scheme works only if there are other (at least one) cooperative or preemptive threads that can run while the wake-up thread and the thread making the I/O call are stopped. It's highly recommended that you never stop the main thread; this will keep you safe for doing thread-blocking I/O with a thread making an I/O call and a wake-up thread. Keeping the main thread ready also keeps the Macintosh user interface active.

To summarize the preceding steps: The cooperative thread doing the I/O creates a cooperative wake-up thread (in the stopped state), makes an asynchronous I/O call, and tells the Thread Manager to put itself in the stopped state. The completion routine fires off any time after the I/O call and makes a call to mark the wake-up thread as ready. Once the thread that made the I/O call goes to sleep, rescheduling occurs and, if the wake-up thread is ready, it's made eligible for running. Normal thread scheduling occurs until the wake-up thread is run. The only thing the wake-up thread needs to do is tell the Thread Manager to set the state of the thread making the I/O call to ready and return. Rescheduling will occur and the thread that made the I/O call will continue on its merry way as if nothing had happened. This process is illustrated in Figure 3.

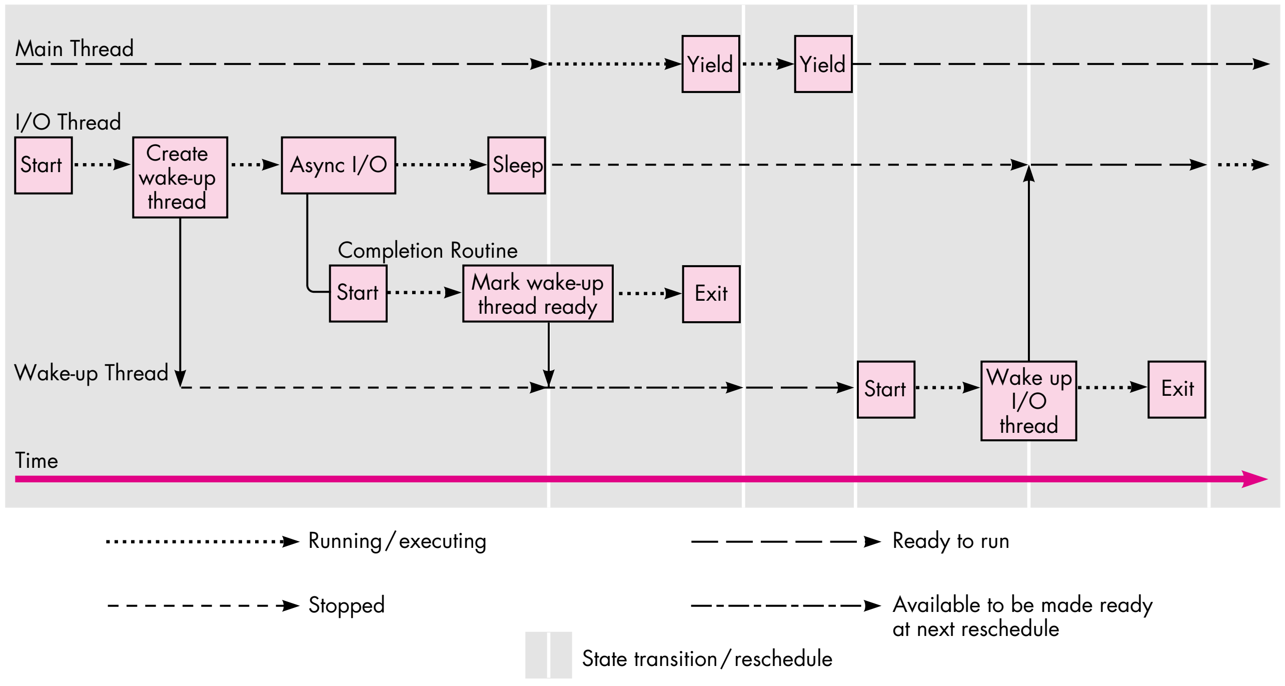


Figure 3
Timeline State Flow Diagram for Threaded I/O

I/O THREAD BLOCKING EXAMPLE

Now let's see how the preceding solution looks in code. We begin by setting up housekeeping and doing some administrative stuff. The extended parameter block structure is used to store the parameter block used by the file system as well as data used by the completion routine. The first element of the structure must be the file system parameter block, as this is the address passed into the file system call. The next two parameters are used by the completion routine to get the application task reference and thread ID of the thread to wake up. These values are stored here because completion routines can't access the application globals when they're called.

To finish up the housekeeping, we define the routine prototypes and the inline routine GetPBPtr. Completion routines are passed the address of the parameter block in register A0, so the GetPBPtr routine is needed to retrieve the parameter block.

```
struct ExtendedParamBlk {
   /* PB must be first so that the file system can get the data. */
   ParamBlockRec  pb;
   ThreadTaskRef  theAppTask;
   ThreadID       theThread;
   };
typedef struct ExtendedParamBlk ExtendedParamBlk;
typedef ExtendedParamBlk *ExtendedParamBlkPtr;

/* Routine prototypes. */
pascal void IOExampleThread (void);
pascal void WakeUpThread (ThreadID threadToWake);
void MyCompletionRoutine (void);

/* Completion routines are called with register A0 pointing to */
/* the parameter block. */
pascal ExtendedParamBlkPtr GetPBPtr(void) = {0x2E88};
                                 /* move.l a0, (sp) */
```

This is a routine in the main thread that creates a thread that makes an I/O call:

```
void KickOffAnIOThread (void)
{
   ThreadID    newCoopID;
   OSErr       theError;

   theError = NewThread(kCooperativeThread,
      (ThreadEntryProcPtr)IOExampleThread, nil, kDefaultThreadStackSize,
      kNoCreationOptions, nil, &newCoopID);
   if (theError)
      DebugStr("\p Could not make cooperative I/O thread");
   /* Return and let the I/O thread do its thing! */
}
```

Below is the code for the thread that makes the I/O call — IOExampleThread. It makes an asynchronous I/O call with a completion routine that will wake up the wake-up thread, which then wakes up IOExampleThread.

```
pascal void IOExampleThread (void)
{
   ThreadID            wakeupThreadID, meThreadID;
   ThreadTaskRef       theAppRef;
```

For more information on completion routines, parameter blocks, and calling routines asynchronously in general, see "Asynchronous Routines on the Macintosh" in *develop* Issue 13.•

```
   ExtendedParamBlk     myAsyncPB;
   OSErr                theError, theIOResult;

   /* Get the ID of IOExampleThread. */
   theError = GetCurrentThreadID(&meThreadID);
   if (theError != noErr)
      DebugStr("\pFailed to get the current thread ID");

   /* Get the application's task reference. */
   theError = GetThreadCurrentTaskRef(&theAppRef);
   if (theError != noErr)
      DebugStr("\Could not get our task ref");

   /* Create a wake-up thread. */
   theError = NewThread(kCooperativeThread,
      (ThreadEntryProcPtr)WakeUpThread, (void*)meThreadID,
      kDefaultThreadStackSize, kNewSuspend, nil, &wakeupThreadID);
   if (theError != noErr)
      DebugStr("\pFailed to create a cooperative thread");
```

Here's where you prepare for the I/O call — a simple asynchronous flush volume command. Notice how we set the address of the completion routine. We also set up the extended data needed by the completion routine — the thread ID of the wake-up thread and the application's task reference.

```
   myAsyncPB.pb.ioParam.ioCompletion = (ProcPtr)MyCompletionRoutine;
   myAsyncPB.pb.ioParam.ioResult = 0;     /* Initialize the result. */
   myAsyncPB.pb.ioParam.ioNamePtr = nil;  /* No name used here. */
   myAsyncPB.pb.ioParam.ioVRefNum = -1;   /* The boot drive. */
   myAsyncPB.theThread = wakeupThreadID;
   myAsyncPB.theAppTask = theAppRef;
```

IOExampleThread makes the I/O call and then calls SetThreadState to put itself to sleep. The first two parameters to SetThreadState indicate the thread to set, which is IOExampleThread, and the state to set it to — stopped. The kNoThreadID parameter indicates that any ready thread can run next.

```
   PBFlushVol((ParmBlkPtr)&myAsyncPB, async);
   theError = SetThreadState(kCurrentThreadID, kStoppedThreadState,
      kNoThreadID);
   if (theError != noErr)
      DebugStr ("\pFailed to put ourselves to sleep");
```

At this point, IOExampleThread is sleeping, but other threads are running (including the main thread, because it's not nice to put it in the stopped state). Meanwhile, thread scheduling is taking place, so when the completion routine executes and tells the scheduler to place the wake-up thread in the ready queue, the wake-up thread can get scheduled to execute. (Make sure the main thread doesn't quit the application with the asynchronous I/O pending. You could run into problems when the completion routine tries to run but no longer exists because the application is gone.)

Next the completion routine fires off and tells the Thread Manager to ready the wake-up thread. The wake-up thread eventually runs and tells the Thread Manager to ready the I/O thread. Then the I/O thread awakes and continues running as if nothing happened, continuing with the rest of the code.

```
   theIOResult = myAsyncPB.pb.ioParam.ioResult;
   . . .
}

void MyCompletionRoutine (void)
{
   ExtendedParamBlkPtr     myAsyncPBPtr;
   ThreadTaskRef           theAppTaskRef;
   ThreadID                theThreadID;
   ThreadState             theThreadState;
   OSErr                   theError;

   /* Get the parameter block. */
   myAsyncPBPtr = GetPBPtr();

   /* Get the data. */
   theAppTaskRef = myAsyncPBPtr->theAppTask;
   theThreadID = myAsyncPBPtr->theThread;

   /* See if the thread is stopped yet - just to be sure. */
   theError = GetThreadStateGivenTaskRef(theAppTaskRef, theThreadID,
      &theThreadState);
   /* If we can get the thread state, go for it! */
   if (theError == noErr)
      {
      /* If it's awake, we have problems. */
      if (theThreadState != kStoppedThreadState)
         DebugStr("\pWake-up thread is in the wrong state!");
      /* Should be sleeping, mark it for wake up! */
      else
         SetThreadReadyGivenTaskRef(theAppTaskRef, theThreadID);
      }
}
```

The wake-up thread eventually begins execution and has only one job — to wake up the thread that made the I/O call.

```
pascal void WakeUpThread (ThreadID threadToWake)
{
   OSErr    theError;

   /* Wake up the specified thread. */
   theError = SetThreadState(threadToWake, kReadyThreadState,
      kNoThreadID);
   if (theError != noErr)
      DebugStr("\pFailed to wake our thread");

   /* We've done our deed, so just return quietly and let it run. */
}
```

I/O WITH PREEMPTIVE THREADS

Things get a little tricky when the thread making the I/O call is preemptive — not to mention that you may not be allowed to make certain asynchronous I/O calls from preemptive threads (you know, the interrupt routine rules and all that). To solve the problem, all you do is force your preemptive thread to look like a cooperative thread. Before you make the asynchronous I/O call, you need to begin a critical section with the ThreadBeginCritical routine. Also, since you can't create threads from preemptive threads (again, the interrupt routine rules), you need to get the wake-up thread from the thread pool you created earlier (you did create one, didn't you?). After the I/O call, call SetThreadStateEndCritical rather than SetThreadState; this puts the thread in the stopped state, ends the critical section to allow preemptive scheduling, and forces a reschedule all at the same time. Then, when the wake-up thread gets run, it knows that the thread that made the I/O call must be sleeping.

FINAL THOUGHTS

Here are some final thoughts before you dive in and start programming with the Thread Manager:

- Remember to always preload the code segments that will be used by preemptive threads. Since you're not allowed to move memory during preemptive thread time, loading a code segment would definitely cause a problem.
- Make sure you call MaxApplZone during the application's initialization, especially before you start creating threads. The application's heap won't grow correctly from other cooperative threads, so you must fully grow your heap before any other thread allocates memory.

- When dealing with cooperative and preemptive threads in the same application, make sure you maintain data coherency between your threads. It's very possible that a preemptive thread could change shared data out from underneath a cooperative thread that's using it. A simple way to maintain data coherency is to use critical sections.
- Be careful calling WaitNextEvent with a large sleep value when there are threads that need time. Threads get time to run only when your application gets time. Putting the application to sleep with WaitNextEvent means that your threads sleep, too.
- Be very careful when quitting an application with suspended threads that will be awakened by an interrupt service routine. This can be a problem in some cases. The Thread Manager provides a thread termination procedure that's called when a thread is about to be disposed of — such as when the application quits — to handle any cleanup needed before final application termination.

We urge you to use the Thread Manager because it will be integrated directly into the system software, and it will be upwardly compatible with the new RISC and OS directions Apple is taking. If you start taking advantage of the Thread Manager now, you'll only be more ahead in the future.

THANKS TO OUR TECHNICAL REVIEWERS
Dave Falkenburg, Michael Gough, C. K. Haun, John Yen•

VIEW FROM THE LEDGE

TAO JONES

Before we get rolling here, I have a confession to make to those of you who have read this column in the last two issues. I feel I owe it to you for being so faithful. Due to lead times, publishing deadlines, and attempts to appear organized, *develop* authors actually write their MacNuggets of wisdom *months* before you read them. You've already had a chance to be in a New Year's Day fight with your significant other, yet I'm writing this before Hallowe'en.

How do I put this? I've been lying to you. The questions in my earlier columns were fakes. I made them up for my own self-aggrandizement. Which is bad enough, but I also gave myself the heartwarmingly collectible gift that's supposed to be given to the dedicated readers who write in. Like Richard Nixon in the seventies, I guess the only defense I have is: I misspoke myself.

But that's the past, and now my head reels because I've received *four* letters. (Well, actually I received three and *develop*'s editor got a piece of hate mail.) Now I know what it's like to be 1/150,000th of "Dear Abby."

Dear Tao,

Help! We have a power-hungry team member who is making the rest of the team thoroughly miserable! What can we do? Mutiny has crossed our collective minds, but after reading Mutiny on the Bounty *none of us fancies suffering the fate of Mr. Christian.*

Desperate

Dear Desperate,

In general you'll find that the world is in turmoil for no reason other than intolerance. If people simply were more tolerant of each other's beliefs, values, customs, and driving skills, the world would be a much more hospitable place. Typically you should try to live in as much harmony as possible with your office workers in an effort to spread peace and happiness in the world. This, however, is not one of those times.

You may be surprised to learn that the answer to your problem was at your fingertips; you just failed to dig far enough for it. All you needed to do was ask yourself a simple question: What *was* the fate of Fletcher Christian?

Historians have actually tried to cover up the *real* reason there was mutiny: it had nothing to do with ill-tempered leadership. Rather, Mr. Christian couldn't stand being in an island-sized sauna with the obese Captain Bligh any longer. To remedy the situation, Mr. Christian decided to send Bligh on a crash diet the old-fashioned way: by making him row across 3500 miles of open ocean.

Shortly after Bligh left, Mr. Christian got out his Apple I computer with cassette tape backup (remember, this was a *long* time ago) and discovered that, by Jove, Bligh might actually *make* it. The last thing Mr. Christian wanted was to witness a size-6 Bligh parade his new physique around the island. So, he gathered up a bunch of Tahitian babes, sailed over to Pitcairn Island, burned the Bounty, and lived the rest of his life sipping coconut juice in a tropical paradise. The only penalty was that he would never be able to go back to a country where they serve a dish called "Spotted Dick" for dessert. To say it was a fair trade is a gross understatement.

TAO JONES was stunned when he discovered that the actual lyrics to The Beatles' "Let It Be," were, "and in my hour of darkness, she is standing right in front of me." He thought the line was, "and in my *oblidoplous*. . ." and spent the next ten years trying to figure out what an oblidoplous was.•

Tao Index: Roughly 50% of the lawyers in the world graduated in the bottom half of their class.•

Taking a cue from Mr. Christian, I'd say the following actions are in order: First, get a detailed plan of your building(s) and find the spot that's the furthest from where the mutiny will actually take place; this will be your equivalent of Pitcairn Island. Due to twentieth-century building layout, it's very possible that you may have to choose someplace like a boiler room. This gives you the tropical climate by default, but I'd add some sand and a few posters just to spice the place up a bit. Don't forget to stock enough coconut milk to last until your management turns its attention to some other crisis. Two weeks worth is probably more than enough.

Then, mutiny to your heart's content. Start by issuing a new org chart, and when your troublemaker comes to protest, just do whatever seems to be most natural. Be as loud and obnoxious as possible, and don't forget to throw in lines like "You call yourself a ship's Captain?" and "From now on you get your own breadfruit trees!" Burning things to the ground is optional. I wouldn't recommend it if your paycheck is important to you.

There's a good chance you'll be set for life. Just be sure you don't get so carried away that you start sending postcards to your colleagues from your island retreat. Nothing gives away a hiding spot like a postmark.

Dear Tao,

Every time I get up quickly I become dizzy. Everyone thinks I have a drinking problem. What can I do?

Spinning in Sacramento

Dear Spinning,

You're in a situation that has very serious physical and sociological implications and needs immediate attention. First and foremost, you should get yourself to a doctor. You may have something as complicated and life threatening as transient ischemic attack, or you may just be a ditz. Only a licensed physician will be able to tell for sure. By no means should you rely on self-diagnosis from watching reruns of "Marcus Welby, M.D." Remember, the guy who played Dr. Kiley tried that and he ended up in *The Amityville Horror.*

Whatever you do, if you go to a university to get checked, be *certain* the person you're talking to is a Medical Doctor. You can't swing a dead cat in that environment without hitting someone who is all too willing to be called "Doctor" yet thinks that "throat culture" has something to do with opera. These people have the title "Doctor of Philosophy," and their specialty is to heal problems with philosophical ideals. If you believe your dizziness is caused by an inability to understand dialectic materialism, this type of doctor is perfect; otherwise, steer clear.

Unfortunately, getting fixed physically won't heal the seeping sociological wound that has opened by everyone thinking you're intoxicated. To tackle this part, first you should decide whether you deserve any time off from work. If you think you do, tell your boss you believe you have a substance abuse problem, and ask to have at least a month off to get yourself treated. If you happen to acquire a tan while you're gone, say that you had to have minor UV therapy for jaundice.

If you don't want the time off, or you want to squelch those rumors once and for all, simply stand up in your next company meeting and say something like, "I know many of you think I have a drinking problem due to my dizzy spells. This simply is not true. I just have a very strange habit of having my head reel whenever I get around a corporate back-stabbing weasel."

RECOMMENDED READING AND LISTENING

- *The Red Couch* by Clarke, Moon, and Wackerbarth. Surprisingly, it's lots of pictures of a red couch.
- *War and Peace* by Leo Tolstoy. Good book. Great monitor stand.
- *Godzilla*, on Star Child records. The original soundtrack; makes all other music sound *great.*

Have you seen this polar bear cub?•

Tao needs questions to keep from hoarding the freebies he is supposed to be sending out to you, the devoted reader. Send your queries regarding all aspects of office survival, or just that funny little thing we call "life," to AppleLink DEVELOP, and there's a possibility that you'll end up even cooler than you are now.•

THE ZEN OF WINDOW ZOOMING

Window zooming is a feature of the Macintosh user interface that's rarely implemented correctly. Because a lot of calculation and pixel-tweaking is required to achieve the "perfect" window zoom, few applications go through the effort to properly zoom their document windows. This article discusses proper zooming etiquette and provides a routine that deals with all the details of zooming windows.

DEAN YU

About once a year, I go on a tirade about how few programs zoom their windows properly. Most programs zoom to the full size of the main screen; other applications make the window only as big as they need to but still move it to the main screen, even if the window is on another screen. The System 7 Finder comes close to making me happy, but every once in a while it zips a window to the main screen for reasons known only to the Finder engineers.

As I was writing some application code recently, I took the opportunity to make it zoom windows the way I wish all other programs would zoom windows. The resulting code (which is on this issue's CD) is the basis for this article. The zooming behavior that this code implements reflects all the documentation Apple has ever published about window zooming, from the old Human Interface notes to the Window Manager chapter in *Inside Macintosh: Macintosh Toolbox Essentials.*

THE ETIQUETTE OF ZOOMING

First let's look at the subtle effects of a user's actions on how a window should be zoomed. After some basic definitions, we'll go over a few rules that govern zooming behavior. If you couldn't care less about these preliminaries, you can skip to the section "The Zooming Code" (however, if you do this, you'll hurt the author's feelings, since he spent a perfect Saturday afternoon indoors to write this article).

THE STATES OF A WINDOW

A window is zoomed between two states, the *user state* and the *standard state*. The user state is any size and position in which the user can place the window on the desktop.

DEAN YU recently went on a cross-country road trip in his new car. In retrospect, he decided that the time off wasn't worth the emotional trauma of being pelted by marble-sized hail in Cheyenne, Wyoming, being salt-blasted in the Utah salt flats, being nearly blown off the road as an 18-wheeler passed him at 95 miles an hour in the dead of night, and having his windshield chipped by gravel falling off a dump truck. Dean didn't even get the satisfaction of finding out exactly how fast he can drive without getting pulled over for speeding in Nebraska. Sympathy notes and donations for getting his car repaired can be sent to Dean in care of *develop*.•

The Window Manager updates the user state when it calls a WDEF to recalculate a window's regions.

The standard state of a window is defined to be the size that can best display the data contained in the window. For example, in word processing applications the standard state of a document window would most likely be the size of a printed page. For some types of windows the standard state depends on the window's contents and so is determined dynamically by the application when the user zooms the window. In the Finder, for instance, the standard state of a window in an icon view would be the smallest size that can display all the icons in that window. The position of a window in its standard state varies depending on the position of the user state when the window was zoomed and on other factors, as explained later.

Figure 1 shows the user state and standard state of a Finder window displayed in the "small icon" view.

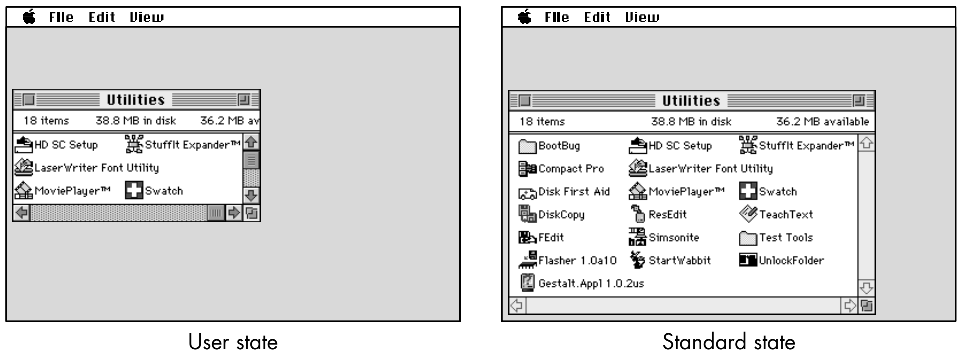


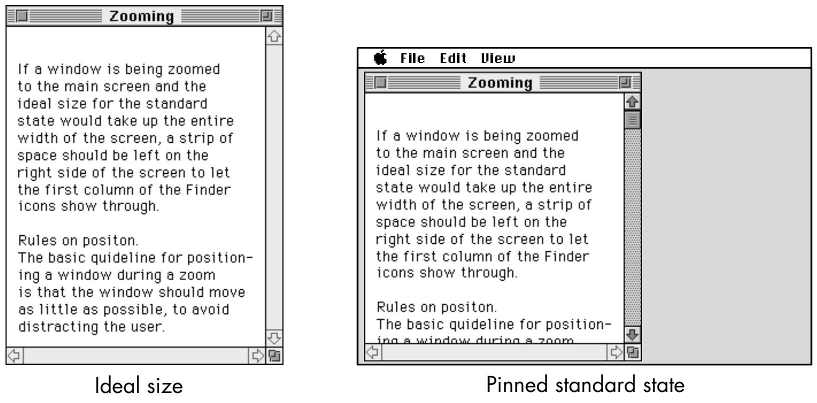
Figure 1
The User and Standard States of a Window

RULES OF THE ROAD

In addition, there are a few rules that govern how a window should be zoomed in different situations. These rules can be divided into two categories: rules on size and rules on position.

Rules on size. Although the standard state of a window is defined to be the best size for displaying the window's contents, this state is actually constrained by the size of the screen to which the window is being zoomed. If the ideal size for the standard state is larger than the destination screen, the window should be pinned to the size of the screen, minus some slop pixels. For example, Figure 2 shows the height of the standard state being pinned on the main screen. (Note that space was also left for the menu bar.)

A lot of people think the Window Manager has some magic value stashed away in a dark corner that tells it whether a window is in its user state or its standard state. In reality, it's not that sophisticated. It's actually the WDEF that does all the work, since it's the thing that really needs to know which state a window is in. To make this decision, the standard document WDEF takes a window's portRect, converts it to global coordinates, and then compares it with the user and standard state rectangles to determine which state the window is in.•



If a window is being zoomed
to the main screen and the
ideal size for the standard
state would take up the entire
width of the screen, a strip of
space should be left on the
right side of the screen to let
the first column of the Finder
icons show through.

Rules on positon.
The basic guideline for position-
ing a window during a zoom
is that the window should move
as little as possible, to avoid
distracting the user.

If a window is being zoomed
to the main screen and the
ideal size for the standard
state would take up the entire
width of the screen, a strip of
space should be left on the
right side of the screen to let
the first column of the Finder
icons show through.

Rules on positon.
The basic guideline for position-
ing a window during a zoom

Figure 2
Pinning to the Screen Size

If a window is being zoomed to the main screen and the ideal size for the standard state would take up the entire width of the screen, a strip of space should be left on the right side of the screen to let the first column of Finder icons show through.

Rules on position. The basic guideline for positioning a window during a zoom is that the window should move as little as possible, to avoid distracting the user.

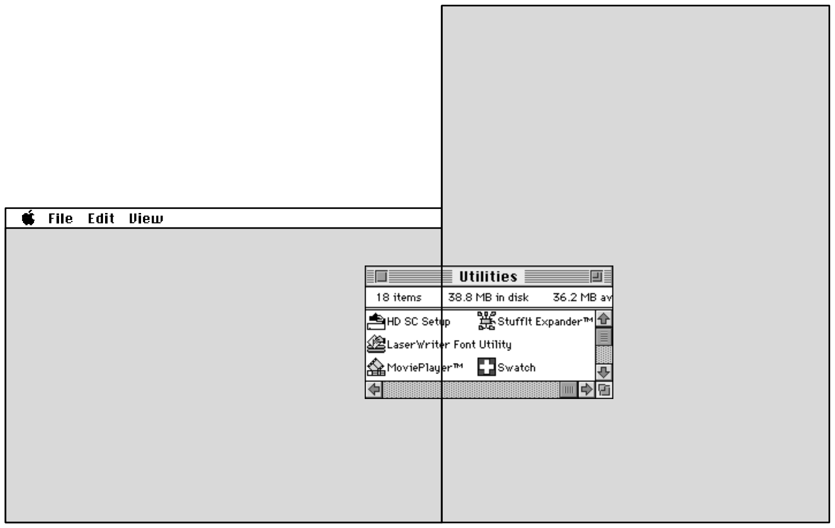
A window in its standard state should be positioned so that it's entirely on one screen. If a window straddles more than one screen in the user state and is subsequently zoomed to the standard state, it should be zoomed to the screen that contains the largest portion of the window's content region. (See Figure 3.)

When a window is zoomed from the user state to the standard state, it should be anchored at its current top left corner if possible. If the standard state size will fit on the screen without moving the window, the window can simply be resized. If the standard state of the window cannot fit with the top left corner anchored, the window should be "nudged" so that the parts that were off the screen fall just on screen. (See Figure 4.)

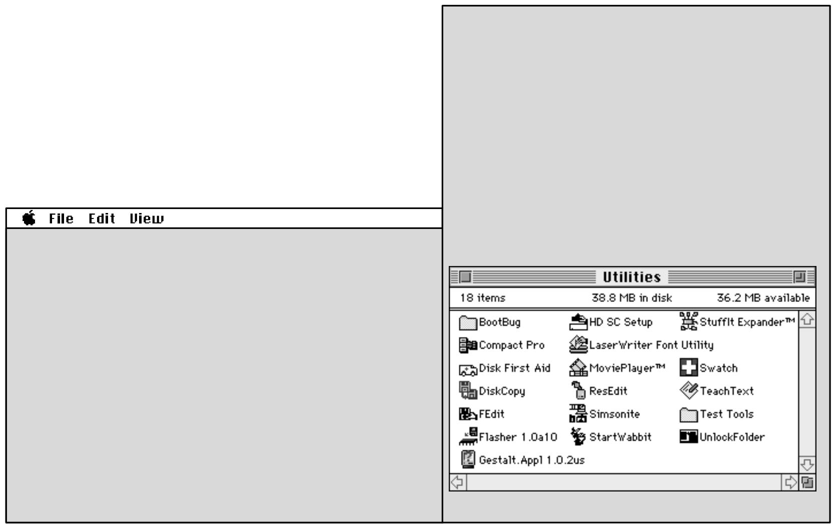
THE ZOOMING CODE

This section goes through the window zooming code chunk by chunk, discussing the logic behind each step.

ZoomTheWindow is the entry point to the window zooming code. It determines the best screen to zoom the window to, and nudges the window into position in case part of it falls off the edge of the screen. Applications should call this routine instead of calling the Toolbox routine ZoomWindow directly.

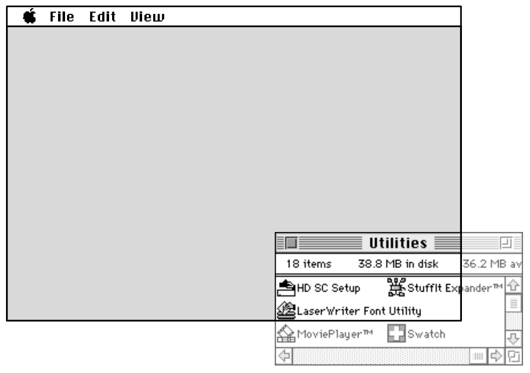


User state

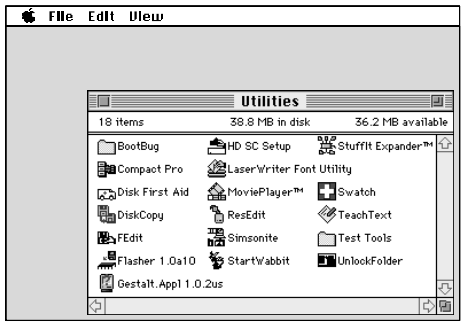


Standard state

Figure 3
Zooming to the Best Screen



User state



Standard state

Figure 4
Nudge Zooming

ZoomTheWindow has the following prototype:

```
void ZoomTheWindow(WindowPeek theWindow, short zoomState,
                   CalcIdealDocumentSizeProcPtr calcRoutine);
```

The first two parameters, theWindow and zoomState, are identical to the first two parameters of ZoomWindow. The last parameter, calcRoutine, is an application-supplied callback routine that calculates the ideal size for the window without taking the user's screen configuration into consideration. The prototype of the CalcIdealDocumentSizeProcPtr function type is as follows:

```
typedef void (*CalcIdealDocumentSizeProcPtr) (WindowPtr theWindow,
                                              Rect *idealContentSize);
```

Given the window to be zoomed, the callback routine returns (in local coordinates) the ideal rectangle for the window in the idealContentSize parameter. The window will be the current graphics port when the callback routine is invoked.

ZoomTheWindow calls two utility routines, CalculateWindowAreaOnScreen and CalculateOffsetAmount. CalculateWindowAreaOnScreen calculates the area of a window on a screen. The screen that contains the largest portion of the window is the screen that the window will be zoomed to. If ZoomTheWindow determines that anchoring the window at its current top left corner will result in part of the window lying off the screen, it calls CalculateOffsetAmount to find out how many pixels the window needs to be nudged so that it's entirely on the screen. These utility routines are described in detail following the discussion of ZoomTheWindow below.

THE ZOOMDATA STRUCTURE

The ZoomData structure is used by ZoomTheWindow to hold information about the screen the window should be zoomed to. ZoomTheWindow uses DeviceLoop to find the screen containing the largest portion of the window. The DeviceLoop drawing procedure updates the ZoomData structure as DeviceLoop calls it for each active screen device.

```
struct ZoomData {
   GDHandle       screenWithLargestPartOfWindow;
   unsigned long  largestArea;
   Rect           windowBounds;
};
typedef struct ZoomData ZoomData, *ZoomDataPtr;
```

The screenWithLargestPartOfWindow field is a handle to the screen device that the window should be zoomed to. The largestArea field holds the area of the largest portion of the window encountered so far, as DeviceLoop iterates through the screens. The windowBounds field is the portion of the window that's currently visible on the desktop.

THE ZOOMTHEWINDOW ROUTINE

About 90% of the code in ZoomTheWindow executes only when the window is to be zoomed to the standard state. The routine starts by setting up the current graphics port and getting some frequently used fields out of the window record.

```
void ZoomTheWindow(WindowPeek theWindow, short zoomState,
                     CalcIdealDocumentSizeProcPtr calcRoutine)
{
   ZoomData    zoomData;
   Rect        newStandardRect, scratchRect, screenRect, portRect;
   Rect        contentRegionBoundingBox, structureRegionBoundingBox;
   Rect        deviceLoopRect;
   GrafPtr     currentPort;
   RgnHandle   scratchRegion, contentRegion, structureRegion;
   GDHandle    mainDevice;
   short       horizontalAmountOffScreen, verticalAmountOffScreen;
   short       windowFrameTopSize, windowFrameLeftSize;
   short       windowFrameRightSize, windowFrameBottomSize;

   GetPort(&currentPort);
   SetPort((WindowPtr) theWindow);
   contentRegion = GetWindowContentRegion(theWindow);
   structureRegion = GetWindowStructureRegion(theWindow);
   GetWindowPortRect(theWindow, &portRect);
   contentRegionBoundingBox = (**contentRegion).rgnBBox;
```

```
structureRegionBoundingBox = (**structureRegion).rgnBBox;
windowFrameTopSize = contentRegionBoundingBox.top -
                        structureRegionBoundingBox.top;
windowFrameLeftSize = contentRegionBoundingBox.left -
                        structureRegionBoundingBox.left;
windowFrameRightSize = structureRegionBoundingBox.right -
                        contentRegionBoundingBox.right;
windowFrameBottomSize = structureRegionBoundingBox.bottom -
                        contentRegionBoundingBox.bottom;
```

Determining the proper screen. The code then determines which screen contains the largest portion of the window, as follows:

```
// If the window is being zoomed to the standard state, calculate the
// best size to display the window's information.
mainDevice = GetMainDevice();
if (zoomState == inZoomOut) {
   zoomData.screenWithLargestPartOfWindow = mainDevice;
   zoomData.largestArea = 0;

   // Get the portion of the window that's on the desktop.
   scratchRegion = NewRgn();
   SectRgn(GetGrayRgn(), contentRegion, scratchRegion);
   if (EmptyRgn(scratchRegion))
      zoomData.windowBounds = structureRegionBoundingBox;
   else
      zoomData.windowBounds = contentRegionBoundingBox;

   // Use DeviceLoop to walk through all the active screens to find the
   // one with the largest portion of the zoomed window.
   deviceLoopRect = zoomData.windowBounds;
   GlobalToLocal((Point *)&deviceLoopRect);
   GlobalToLocal((Point *)&deviceLoopRect.bottom);
   RectRgn(scratchRegion, &deviceLoopRect);
   DeviceLoop(scratchRegion, &CalcWindowAreaOnScreen, (long) &zoomData,
            (DeviceLoopFlags) singleDevices);
   DisposeRgn(scratchRegion);
   screenRect = (**(zoomData.screenWithLargestPartOfWindow)).gdRect;

   // If the window will be zoomed to the main screen, change the top
   // of the usable screen area so that the window's title bar won't be
   // placed under the menu bar.
   if (zoomData.screenWithLargestPartOfWindow == mainDevice)
      screenRect.top += GetMBarHeight();
```

ZoomTheWindow sets up default values in the ZoomData structure so that the window will zoom to the main screen by default. Normally, the content area of the window is used to determine the area of the window that's on each screen. However, if the user has the window positioned such that only the title bar (or a portion of the title bar) is visible on the desktop, the structure region of the window is used to determine the screen that contains the largest portion of the title bar.

One of the little-known facts about the universe is that inspiration actually comes from subatomic particles flying around in deep space. These particles occasionally hit a sentient brain, resulting in a flash of inspiration. One of these events resulted in the use of DeviceLoop to iterate through the list of active screen devices. (Of course, this means that the zooming code requires System 7; for a System 6 alternative to DeviceLoop, see "Graphical Truffles: Multiple Screens Revealed" in *develop* Issue 10.) The CalcWindowAreaOnScreen routine is the DeviceLoop drawing procedure. The ZoomData structure is passed to DeviceLoop as the userData value, and the DeviceLoop flags are set so that DeviceLoop will call its drawing procedure for each screen device.

Determining the ideal size for the window. After determining the proper screen to zoom to, the application's callback routine is called to get back the ideal content size for the window. This rectangle is then anchored at the window's current top left corner and expanded to include the window frame.

```
// Figure out the perfect size for the window as if we had an
// infinitely large screen.
(*calcRoutine)((WindowPtr) theWindow, &newStandardRect);

// Anchor the new rectangle at the current top left corner of the
// window.
OffsetRect(&newStandardRect, -newStandardRect.left,
            -newStandardRect.top);
OffsetRect(&newStandardRect, contentRegionBoundingBox.left,
            contentRegionBoundingBox.top);

// Add the window frame to the ideal content rect.
newStandardRect.top -= windowFrameTopSize;
newStandardRect.left -= windowFrameLeftSize;
newStandardRect.right += windowFrameRightSize;
newStandardRect.bottom += windowFrameBottomSize;
```

Fitting the ideal size onto the screen. This is the tedious part of the code. At this point, newStandardRect holds the ideal size for the window being zoomed. Since a window in its standard state must be entirely on one screen, we ensure that the window fits on the screen, maintaining its ideal size if possible.

If the structure region is always used to calculate the screen containing most of the window, there's actually a scenario in which the code will zoom to the wrong screen. If the user has a multiscreen setup and places a window so that it sits across two screens, with only the title bar showing on one screen but a sliver of the content region showing on the other, and there's more title bar than content showing, the window will zoom to the first screen. However, since there is content area on the second screen, the window should zoom to that screen.•

```
// If the new rectangle falls off the edge of the screen, nudge it
// so that it's just on the screen. CalculateOffsetAmount determines
// how much of the window is off the screen.
SectRect(&newStandardRect, &screenRect, &scratchRect);
if (!EqualRect(&newStandardRect, &scratchRect)) {
   horizontalAmountOffScreen = CalculateOffsetAmount(
                     newStandardRect.left, newStandardRect.right,
                     scratchRect.left, scratchRect.right,
                     screenRect.left, screenRect.right);
   verticalAmountOffScreen = CalculateOffsetAmount(
                     newStandardRect.top, newStandardRect.bottom,
                     scratchRect.top, scratchRect.bottom,
                     screenRect.top, screenRect.bottom);
   OffsetRect(&newStandardRect, horizontalAmountOffScreen,
               verticalAmountOffScreen);
}

// If we're still falling off the edge of the screen, the perfect
// size is larger than the screen, so shrink the standard size.
SectRect(&newStandardRect, &screenRect, &scratchRect);
if (!EqualRect(&newStandardRect, &scratchRect)) {
   // First shrink the width. If the window is wider than the screen
   // it's being zoomed to, just pin the standard rectangle to the
   // edges of the screen, leaving some slop; otherwise, we know we
   // just nudged the window into position, so do nothing.
   if ((newStandardRect.right - newStandardRect.left) >
         (screenRect.right - screenRect.left)) {
      newStandardRect.left = screenRect.left + kNudgeSlop;
      newStandardRect.right = screenRect.right - kNudgeSlop;

      if ((zoomData.screenWithLargestPartOfWindow == mainDevice) &&
         (newStandardRect.right > (screenRect.right - kIconSpace)))
         newStandardRect.right = screenRect.right - kIconSpace;
   }

   // Move in the top of the window. As with the width of the
   // window, do nothing unless the window is taller than the
   // height of the screen.
   if ((newStandardRect.bottom - newStandardRect.top) >
         (screenRect.bottom - screenRect.top)) {
      newStandardRect.top = screenRect.top + kNudgeSlop;
      newStandardRect.bottom = screenRect.bottom - kNudgeSlop;
   }
}
```

```
      // We've got the best possible window position. Remove the
      // frame, slam it into the WStateData record and let ZoomWindow
      // take care of the rest.
      newStandardRect.top += windowFrameTopSize;
      newStandardRect.left += windowFrameLeftSize;
      newStandardRect.right -= windowFrameRightSize;
      newStandardRect.bottom -= windowFrameBottomSize;
      SetWindowStandardState(theWindow, &newStandardRect);
   }  // if (zoomState == inZoomOut)
   else
      GetWindowUserState(theWindow, &newStandardRect);
```

We call CalculateOffsetAmount to determine how much to nudge the window if it falls off the edge of the screen in its ideal size. After nudging the window, we double check to see if the window is entirely on the screen. If it still isn't, that means that the ideal size of the window is larger than the screen that it's zooming to, so the window has to be shrunk to fit on the screen. The code shrinks the window so that there's a small area of slop space between the edge of the window and the screen boundary. Additionally, if the screen that the window is zooming to is the main screen, space is left for the menu bar and a column of Finder icons.

After all that, newStandardRect contains the best size for the window for that screen. After we remove the window frame, the stdState field of the WStateData record can be filled with this rectangle.

There's also the simple case of zooming to the user state. Since the Window Manager takes care of keeping the userState field of the WStateData record up to date, that rectangle is easy to get.

Zooming the window. Finally, all that's left to do is to actually zoom the window. One final optimization that can be performed is that if the top left corner of the window hasn't moved, SizeWindow can be called instead of ZoomWindow, reducing the amount of redrawing that needs to be done. The window's clipping region is reset to be the size of the window to ensure that the window's contents are entirely erased before ZoomWindow is called.

```
   // If the window is still anchored at the current top left corner, just
   // resize it.
   if ((newStandardRect.left == contentRegionBoundingBox.left) &&
         (newStandardRect.top == contentRegionBoundingBox.top)) {
      OffsetRect(&newStandardRect, -newStandardRect.left,
                  -newStandardRect.top);
      SizeWindow((WindowPtr) theWindow, newStandardRect.right,
                  newStandardRect.bottom, true);
   }
   else {
      scratchRegion = NewRgn();
      GetClip(scratchRegion);
      ClipRect(&portRect);
      EraseRect(&portRect);
      ZoomWindow((WindowPtr) theWindow, zoomState, false);
      SetClip(scratchRegion);
      DisposeRgn(scratchRegion);
   }
   SetPort(currentPort);
}
```

THE CALCWINDOWAREAONSCREEN ROUTINE

CalcWindowAreaOnScreen, the DeviceLoop drawing procedure for the zooming code, doesn't actually do any drawing, but instead simply calculates the area of the window that's on a screen. If there's more content area on one screen than any of the other screens that have been encountered so far, CalcWindowAreaOnScreen saves the GDHandle of this screen in the ZoomData structure as the potential screen to zoom the window to.

```
pascal void CalcWindowAreaOnScreen(short depth, short deviceFlags,
                                   GDHandle targetDevice, long userData)
{
#pragma unused (depth, deviceFlags)
   ZoomDataPtr    zoomData = (ZoomDataPtr) userData;
   unsigned long  windowAreaOnScreen;
   Rect           windowPortionOnScreen;

   // Find the rectangle that encloses the intersection of the screen and
   // the document window.
   SectRect(&(zoomData->windowBounds), &((**targetDevice).gdRect),
            &windowPortionOnScreen);

   // Offset this rectangle so that its right and bottom are also its
   // width and height.
   OffsetRect(&windowPortionOnScreen, -windowPortionOnScreen.left,
               -windowPortionOnScreen.top);

   // Calculate the area of the part of the window that is on this
   // screen.
   windowAreaOnScreen = windowPortionOnScreen.right *
                        windowPortionOnScreen.bottom;

   // If this is the largest area that has been encountered so far,
   // remember this screen as the potential screen to zoom to.
   if (windowAreaOnScreen > zoomData->largestArea) {
      zoomData->largestArea = windowAreaOnScreen;
      zoomData->screenWithLargestPartOfWindow = targetDevice;
   }
}
```

THE CALCULATEOFFSETAMOUNT ROUTINE

The zooming code calls the CalculateOffsetAmount routine to calculate the number of pixels the window needs to be nudged to be entirely on the screen. This routine works in one dimension at a time, so ZoomTheWindow calls it twice, once for the width of a window and once for the window's height. If CalculateOffsetAmount determines that the window is larger than the screen, it returns 0 for the offset, since the window will be resized later.

```
short CalculateOffsetAmount(short idealStartPoint, short idealEndPoint,
               short idealOnScreenStartPoint, short idealOnScreenEndPoint,
               short screenEdge1, short screenEdge2)
{
   short offsetAmount;

   // Check to see if the window fits on the screen in this dimension.
   if ((idealStartPoint < screenEdge1) && (idealEndPoint > screenEdge2))
      offsetAmount = 0;

   else {
      // Find out how much of the window lies off this screen by
      // subtracting the amount of the window that's on the screen from
      // the size of the entire window in this dimension. If the window is
      // completely off-screen, offset the window so that it's placed just
      // on the screen.
      if ((idealOnScreenStartPoint - idealOnScreenEndPoint) == 0) {
         // See if the window is lying to the left or above the screen.
         if (idealEndPoint < screenEdge1)
            offsetAmount = screenEdge1 - idealStartPoint + kNudgeSlop;
         else
         // Otherwise, it's below or to the right of the screen.
            offsetAmount = screenEdge2 - idealEndPoint - kNudgeSlop;
      }
      else {
         offsetAmount = (idealEndPoint - idealStartPoint) -
                        (idealOnScreenEndPoint - idealOnScreenStartPoint);

         // If we're nudging, add slop pixels.
         if (offsetAmount != 0)
            offsetAmount += kNudgeSlop;
         // Check to see which side of the screen the window was falling
         // off of, so that it can be nudged in the opposite direction.
         if (idealEndPoint > screenEdge2)
            offsetAmount = -offsetAmount;
      }
   }
   return offsetAmount;
}
```

CalculateOffsetAmount determines the nudge amount by calculating the amount of overlap of two line segments. The first line segment, described by the idealStartPoint and idealEndPoint parameters, is the width or height of the window being zoomed. The second line segment, described by the idealOnScreenStartPoint and idealOnScreenEndPoint parameters, is the part of the window's width or height that is on the screen the window will be zoomed to. The number of pixels the window will be nudged is the difference between the lengths of these two line segments, plus some slop. If the length of the second line segment is 0, the window is entirely off the screen that it will be zoomed to. In this case, CalculateOffsetAmount will return the number of pixels the window will have to be nudged to be just on the screen. A third line segment, describing the screen width or height, is used to check whether the window is larger than the screen and to determine the direction to nudge the window.

ZOOMING AWAY

The code presented in this article takes care of most of the work of zooming windows. All your application needs to do is supply the code that determines the ideal size for your windows. Hopefully, many more applications will implement proper zooming behavior in the near future. This will make the people in the offices around me especially happy, since it's one less thing I'll have to complain about.

RECOMMENDED READING

- "DeviceLoop Meets the Interface Designer" by John Powers, *develop* Issue 13. How to use DeviceLoop the correct way.
- *Making It Macintosh: The Macintosh Human Interface Guidelines Companion* (Apple Computer, 1993). Anyone who wants to write programs on the Macintosh should be required to go through this CD first. APDA #R0450LL/A.
- *Sourcery* by Terry Pratchett (Signet, 1989). Find out about inspiration particles, magicons, and other subatomic particles and how they interact with ducks.

THANKS TO OUR TECHNICAL REVIEWERS
C. K. Haun, Elizabeth Moller, Dave Owens,
Craig Prouse•

TEN TIPS FOR GAME DEVELOPERS

BRIGHAM STEVENS

In this column I'll give some general tips that are targeted at game developers but can in fact benefit any Macintosh application. Many of the tips are illustrated in the accompanying sample code (CopyBits ColorKarma) on this issue's CD. If you're considering writing a game for the Macintosh, or you want to improve your existing game or other application, these tips are for you. Here they are at a glance:

1. The Macintosh gaming market is wide open.
2. Bypass QuickDraw wisely.
3. Use CopyBits correctly.
4. Scroll graphics smoothly.
5. Don't synchronize with the VBL interrupt.
6. Use Sound Manager 3.0.
7. Learn when to use (or not use) Apple events.
8. Use the Time Manager.
9. Use the Memory Manager effectively.
10. Use a compatible copy-protection scheme (if any).

THE TIPS IN DETAIL

1. The Macintosh gaming market is wide open.

The Macintosh has infiltrated the homes, offices, and schools of every continent on the planet, and it has matured enough to be ready for entertainment software of all sorts. Hit Macintosh games have sold over 60,000 copies, and users are clamoring for more. If you walk into a software store, though, you'll notice that there isn't a very large selection of Macintosh game and entertainment software. Now is the time to take advantage of the lack of competition and get into the entertainment market with your games.

2. Bypass QuickDraw wisely.

Applications that bypass QuickDraw by accessing video memory directly may not work on future Macintosh platforms. The Macintosh is evolving, and some of the changes may be in the bus architecture, causing applications that write to video memory to break. We're not saying this is going to happen any time soon, but it's a good idea to be ready for it now so that your product will last in the marketplace.

A good compromise is to draw with custom drawing code into an off-screen GWorld, and then use CopyBits to transfer it to the screen. This avoids accessing hardware directly, and will always work.

If you must write directly to the screen, it's important to follow the guidelines set forth in "Graphical Truffles: Writing Directly to the Screen" in *develop* Issue 11, which states that you should always have a QuickDraw version of your code. If your program uses QuickDraw (or QuickDraw GX), it will always be compatible with every future Macintosh platform.

If you do bypass QuickDraw, your application should time the custom drawing code versus QuickDraw at run time, and then choose the fastest routines. This way, the fastest code will be used in cases where your code is running on a system that has an accelerated version of QuickDraw, or perhaps a different CPU altogether.

3. Use CopyBits correctly.

I've heard many developers say that CopyBits is slow, that it can't achieve the frame rates needed to do good games. If you use CopyBits correctly, however, you *can* achieve a high animation frame rate.

BRIGHAM STEVENS (Internet vikingmind@aol.com) Since you last saw Brigham here, he has spoken at the Worldwide Developers Conference on Macintosh game development, moved to San Francisco, jumped out of a perfectly good airplane, become addicted to flight simulators, spent 150 hours in a car with no stereo, tossed his cookies on the steps of the Smithsonian Air and Space Museum in Washington DC, quit Apple to start a game development/consulting company in San Francisco, and become a vampire. You'll see some games from him later on in 1994.•

Understanding all of the factors that affect CopyBits performance is critical to achieving high animation frame rates and still having processor bandwidth left over for the rest of the game. The following tips on CopyBits speed have been collated from many different sources, including Technical Notes, sample code, and other tomes of QuickDraw knowledge. (See also tip 4 below.)

- CopyBits is more efficient with wider images than with tall ones.
- CopyBits is more efficient with rectangular transfers, with no mask region.
- CopyBits is more efficient when the source and destination have the same color table, and even better when the ctSeeds of the color tables are the same. (The accompanying sample code shows how to use the Palette Manager and how to set up the color tables in your application window and off-screen GWorld.)
- CopyBits with a mask region is faster than either CopyMask or CopyDeepMask. Convert your masks to QuickDraw regions and then use CopyBits. (See the CopyBits vs. CopyMask snippet on this issue's CD.)
- CopyBits is faster when the transfer source and destination are long-word aligned, especially on an 68040-based Macintosh.

For more details on these principles, see the Tech Note "Of Time and Space and _CopyBits."

4. Scroll graphics smoothly.

This tip applies to games with scrolling maps, painting programs, and any application that needs to scroll window content quickly and smoothly.

Many Macintosh applications suffer from flickering scrolling, caused by erasing the previous image before drawing the new image. To reduce flicker, you should redraw only the parts of the screen that change; don't erase anything first, unless absolutely necessary. When you're designing your scrolling code or animation engine, the philosophy to adopt is that every pixel should be touched only *once*.

Another technique is to buffer your graphics into an off-screen GWorld: make all changes in the GWorld and then transfer the image to the screen with CopyBits. This can be slower, because the image is drawn twice, but it results in an especially smooth update. An example of a game that uses CopyBits in this way is MacPlay's Out of This World. This game draws into an off-screen GWorld using custom polygon-rendering code (thus ensuring a high frame rate); then, when the image is completely rendered, it's transferred to the screen with CopyBits (ensuring compatibility with future video hardware). For more on this subject, see "Graphical Truffles: Animation at a Glance" in *develop* Issue 12 and "Drawing in GWorlds for Speed and Versatility" in Issue 10.

There's another method that isn't as smooth but uses less memory, and that is to use ScrollRect. ScrollRect was changed in System 7: if you pass nil for the updateRgn parameter of ScrollRect, it won't erase the area that has been uncovered, and you can then use CopyBits in a second step to copy in the new bits. (In System 6, ScrollRect will erase the area you're scrolling out of, causing the screen to flicker more.)

The sample code demonstrates these techniques, showing the tradeoffs between memory footprint and smoothness/apparent speed.

5. Don't synchronize with the VBL interrupt.

Many developers have wanted to synchronize animation with the vertical blanking (VBL) interrupt to eliminate tears when the next frame of animation is drawn before the display hardware has completed the previous frame. It's possible to eliminate tears from small-sized animations, but the overall application will run more slowly because you'll be spending time waiting for the VBL period to start. This results in a much lower animation frame rate, and the application also loses processing power for the rest of the program. Note that QuickTime does not synchronize with the VBL interrupt.

For more on VBL interrupts, see the *Guide to Macintosh Family Hardware,* second edition.•

Another headache to consider with respect to synchronizing with the VBL interrupt is that displays have different refresh rates, and each one's actual VBL period has a different length. This means that for your program to have accurate frame rates on different monitors, you'll have to time the refresh rate of the display you're animating on.

To work around not being able to synchronize with the VBL interrupt, you should try to interleave the animation processing so that you're never updating too many objects at one time. The Time Manager will allow you to break the processing up into separate tasks (see tip 8). If you're getting tears on objects, consider using fewer objects or smaller ones.

6. Use Sound Manager 3.0.

The Macintosh Sound Manager has recently been enhanced (version 3.0). It now can efficiently handle as many sound channels as memory and processor bandwidth can take. This means four channels on a Macintosh LC (which used to handle only one channel) and up to 16 or more on higher-end platforms. As a result, your application can play sound and still have enough CPU bandwidth for other animation and processing. The sample code demonstrates multiple sound channels playing asynchronously while animating an image. Also see "Somewhere in QuickTime: What's New With Sound Manager 3.0" in *develop* Issue 16 and "The Asynchronous Sound Helper" in Issue 11.

Many bugs have been fixed in Sound Manager 3.0. You can now open a sound channel at the start of your program and then continuously use SndPlay to play sounds through it, without disposing of the channel between sounds. Previous versions of the Sound Manager had problems playing sampled sounds like this, so many developers adopted the technique of allocating and disposing of a new sound channel for each sound played.

On the AV Macintosh models, the Sound Manager uses the DSP. This requires the DSP Manager to load a new component every time you open a new channel, and may require disk access. So if you're running with Sound Manager 3.0 you should *not* open and close a sound channel for each sound played; doing so will cause your application to perform less than optimally, especially on the AV models.

See the source code file GameSounds.c, which is part of the CopyBits ColorKarma sample, for an example of a unit that manages asynchronous sound. If Sound Manager 3.0 or later is present, the code opens the sound channels at initialization and closes them when the program quits; otherwise it opens and closes the channels as sounds are played.

Sound Manager 3.0 also adds a new routine, named GetSndHeaderOffset, that makes it easier to use a bufferCmd to play sounds. Using a bufferCmd is faster than using SndPlay. See the sample code on the CD for an example.

Note that the sample code doesn't store the application's A5 register as part of the callback command, so that the interrupt code can set the flag associated with the channel. Instead it just stores a pointer to the flag. This allows the interrupt-time callback to be very small, since it doesn't have to save, set up, and restore A5; it just dereferences the pointer and sets the flag directly. The sample code gives an example of playing a sound asynchronously with a completion callback. I use this technique in just about any interrupt-time callback code I write, including VBL tasks, Time Manager tasks, and Device Manager completion routines.

If you don't use the Sound Manager at all, you're taking an unnecessary compatibility risk. Apple has always recommended against accessing the sound hardware directly. Applications that violate this rule have broken in the past, and they will break again.

7. Learn when to use (or not use) Apple events.

Apple events have simplified interapplication communication, making it easy to add value to your application. A game played against live human players is often more fun than a game played against a

Sound Manager 3.0 is available for licensing, so you can distribute it with your products to ensure that your customers and applications will receive its benefits. You can reach Apple's software licensing department at AppleLink SW.LICENSE.•

computer. Just about every night you'll find some Apple engineers huddled over their computers playing Bolo, Spaceward Ho!, or other network games.

Consider Velocity's Spectre, a network tank game that has been very successful. Spectre doesn't use Apple events; it uses custom DDP (datagram delivery protocol) socket listeners at the lowest level of AppleTalk to achieve high performance. But if your game doesn't require the same level of performance, you may benefit from the ease of use of Apple events.

If you require more performance than Apple events can provide, one option is to use the PPC Toolbox directly, which will allow you to still remain a step removed from direct AppleTalk. See the PPC Toolbox chapter of *Inside Macintosh* Volume VI for more information.

If you require even more performance, you can use AppleTalk Data Stream Protocol (ADSP) directly, or one of the other AppleTalk protocols. ADSP is a higher-level protocol that will allow you to do block transfers and not worry about losing packets and packet order.

It's hard to determine which networking protocol to use ahead of time. From my experience in using Apple events to synchronize animation and events between two Macintosh computers, I would say that if your game is a more than two-player, real-time arcade game, Apple events would probably not be the best solution. If your game is a turn-based strategy-type game, like Spaceward Ho!, RoboSport, Strategic Conquest, and many others, Apple events will work very well for you, no matter how many players are in the game.

For a simple example of using Apple events as a game messaging system, see ZAM 1.a13 on the CD.

8. Use the Time Manager.

The Macintosh Time Manager is very useful for game developers. Animation code often needs a heartbeat, to synchronize the timing and updates of every object. The Time Manager lets you break down your code into discrete tasks that run at a steady rate. This allows you to write modular code that updates smoothly.

One limitation of the Time Manager is that tasks fire at interrupt time, so they can't do much more than set a flag to inform the regular event loop that it's time to do something. The sample code on the CD shows how to place a wrapper around the Time Manager that allows you to execute tasks at non-interrupt time.

9. Use the Memory Manager effectively.

The Macintosh Memory Manager is very flexible, and a boon to most application programmers. However, for the game programmer it can be a performance problem unless it's used wisely. In a game, you should preallocate as much of your memory as possible. If you're using a dynamic object allocation scheme, you should design one that preallocates the objects and keeps track of which ones are in use or not. If you have many allocated blocks in a heap and then request a new one, you could send the Memory Manager into thrashing mode where it will try to move many blocks around to make space. This can cause your animation to be jerky or your whole game to freeze for an instant. So the best thing to do when performance matters is to minimize your use of the Memory Manager.

To minimize Memory Manager use, you should not only allocate as much of your memory up front as possible but also avoid using relocatable blocks unless absolutely necessary. This means avoiding game architectures that rely on the Memory Manager for dynamic object allocation. Definitely allocate your nonrelocatable blocks first, and allocate handles later. This prevents heap fragmentation and avoids sending the Memory Manager into a tailspin.

Be aware that some parts of the Toolbox, like Apple events, expect Memory Manager structures. However, if the rest of the program's memory is allocated wisely to prevent heap fragmentation, even these allocations will happen quickly, with no impact on game performance.

10. Use a compatible copy-protection scheme (if any).

It's a matter of great controversy whether software should be copy-protected at all. No software protection scheme on the Macintosh has ever survived the talented efforts of Macintosh hackers. There's always someone who will defeat your copy protection, no matter how convoluted it may be. There are many who consider such protection a puzzle and a challenge to break, so by putting it in you may be inviting piracy.

But if you do decide you want some level of protection on your game, we strongly recommend against a disk-based protection scheme, which is guaranteed to break your program. Instead, we recommend using one (or a combination) of the methods described here.

One method is to use serial numbers: when the software is installed, ask the user to enter the serial number from the disk label, and then imprint the software with the person's name. Another method involves requiring the user to enter a password from the manual every once in a while. If you do this, it's a nice touch to allow users who send you the registration card to disable the password dialog; once you have the registration card, you can link the customer to a serial number.

Consider also making the following checks: At installation time, use Gestalt to determine the characteristics of the machine you're installed on, and save these to your preferences file. Also, use FindFolder to record the directory ID of the System Folder, which is the same until a new System Folder is created. Every time your application starts up, make the same Gestalt and FindFolder calls to check whether you're running on the same machine; if not, have the user reinstall the software and reenter the serial number, or reenter the password from the manual.

These techniques are the most compatible way to both protect your sales and minimize the kind of frustration customers experience with other password- or disk-based copy protection systems.

ARE YOU GAME?

That's not all! To help Macintosh game developers share tips, tricks, and information, Apple has set up the Game Development Discussion folder on AppleLink (in Developer Support: Developer Talk). This board is read by Macintosh game development companies, other developers, and Apple engineers on a daily basis. Also, there's a folder on this issue's CD, called Game Development, that contains special resources Apple has put together to aid all game developers.

So if you're tired of the scant choices on the Macintosh Entertainment shelf in your local software store, do something about it: write some games!

REFERENCES

- Macintosh Technical Note "Of Time and Space and _CopyBits" (QuickDraw 21).
- "Somewhere in QuickTime: What's New With Sound Manager 3.0" by Jim Reekes, *develop* Issue 16.
- "Graphical Truffles: Animation at a Glance" by Edgar Lee, *develop* Issue 12.
- "The Asynchronous Sound Helper" by Bryan K. ("Beaker") Ressler, and "Graphical Truffles: Writing Directly to the Screen" by Brigham Stevens and Bill Guschwan, *develop* Issue 11.
- "Drawing in GWorlds for Speed and Versatility" by Konstantin Othmer and Mike Reed, *develop* Issue 10.
- *Guide to Macintosh Family Hardware*, 2nd ed. (Addison-Wesley, 1990).
- *Snow Crash,* by Neal Stephenson. A fast-paced book to keep you up late at night. Stephenson says this book was inspired by the original *Macintosh Human Interface Guidelines.* (Imagine what he would have done if he'd had the new improved edition!)

If you create a demo version of a game by commenting out code or imposing a time limit to game play, be aware that these kinds of demos are often hacked into full-blown pirate applications. We strongly suggest that when you create a demo version, you take out all nonessential pieces of code and sufficiently cripple the remaining software (so that a hacker can't simply paste in missing resources to get a full, unprotected copy).•

Thanks to Konstantin Othmer, Jim Reekes, and John Wang for reviewing this column. And special thanks to the people who contributed to the information in this column, including (but not limited to) C. K. Haun, Tony Myles, Craig Fryar, Mike Schlachter, Bill Dugan of Interplay Productions, and the rest of the developers I've worked with over the past couple of years.•

MACINTOSH Q & A

Q *In QuickDraw GX, if the QuickDraw clipping region of a window (set by SetClip(aRgn)) is not rectangular, the shapes drawn to the window's viewPort are sometimes clipped incorrectly. Is this a bug?*

A This is not a bug. QuickDraw GX doesn't honor any of the QuickDraw clipping regions except the one that's connected to a viewPort, created by calling the GXNewWindowViewPort routine.

Here's why: So that QuickDraw GX can draw into viewPorts associated with windows, it patches the Window Manager. When the user moves a window, or when your application calls BeginUpdate and EndUpdate, QuickDraw GX updates the viewPort caches and clip shape. The clip shape is set to the visRgn of the window. However, all other QuickDraw regions are ignored.

If you want to do some smart window clipping, you can. But you can't play directly with the viewPort attached to the window: that viewPort is maintained by the QuickDraw GX system and is therefore off limits to the application. (If you do try to manipulate it, you'll receive an error.) You need to attach a child viewPort to the window's viewPort; then you can convert the QuickDraw region you want to clip to into a GX shape (with the GX Translator) and set the child viewPort's clip shape to this new shape. Your QuickDraw GX shapes will then be clipped correctly.

Q *Calling GXSetStyleRunControls on a layout shape with the "track" field set doesn't always have any visible effect, although the track value is always safely stored in the layout shape and can be retrieved with GXGetStyleRunControls. You can see this in GXWrite by adjusting the track kerning of some Tekton text: that works fine. If you switch to the Hoefler font, however, no track kerning is performed. Can you explain this?*

A When a track kerning value is specified, QuickDraw GX looks to the tables in the font to see how the kerning should be performed. A narrow text face with a large x-height, like Avant Garde, can probably take much tighter track kerning than something like Hoefler Italic can; only the font designer really knows how the kerning should behave. The font designer tells QuickDraw GX how to do track kerning by setting values in the 'trak' table of the font. This table contains one or more "reference points" — values between -2 and 2 — each of which tells how much to adjust space between the glyphs for any number of point sizes. If the specified track kerning value lies between reference points, QuickDraw GX interpolates accordingly.

Hoefler Italic contains only one reference point — "normal" kerning (value 0). It does contain values for several different point sizes, but the font designer

These answers are supplied by the technical gurus in Apple's Developer Support Center. Special thanks to Pete ("Luke") Alexander, Matt Deatherage, Godfrey DiGiorgi, Steve Falkenburg, Mark ("The Red") Harlan, Dave Hersey, Elli Howe, Joseph Maurer, Kevin Mellander, Jim Mensch, and Brigham Stevens for the material in this Q & A column.•

doesn't want the face more tightly or loosely kerned than in his original design. Since there's only one 'trak' point, QuickDraw GX winds up interpolating those entries with themselves and nothing really happens for the values you specify in the run controls. As you've noted, those values are still there, and they would be used if you switched those runs to a different text face, but they're having exactly the effect Jonathan Hoefler wanted — none at all.

Q *We're developing a QuickDraw GX printer driver for a 24-bit continuous-tone device. First, what should halftone fields be set to for a continuous-tone device (are they ignored?). Second, do we have to override GXDefaultPrinter and GXDefaultJob? The ImageWriter example shows building a default colorset inside GXDefaultPrinter.*

A The answer to your first question is that you can stop QuickDraw GX from using the halftone information by setting the gxDontSetHalftone flag in the plane flags field of each plane in the 'rdip' resource. That's it — the halftone fields will be ignored.

The messages GXDefaultPrinter, GXDefaultDesktopPrinter, GXDefaultJob, GXDefaultPaperType, and GXDefaultFormat may be appropriate for your driver to override in order to modify default objects. You don't need to override these messages unless you have a reason to change the default object.

The ImageWriter driver with QuickDraw GX adds a list of view devices that are appropriate for the printer that's being used. Specifically, it adds a 144-dpi black-and-white view device and, if color is available, a 144-dpi color view device. If the driver didn't add these, the default printer view device (24-bit at 72 dpi) would be used. Overrides of a GXDefaultxxx message typically forward the message first to obtain the default object, and then modify that object as needed. So if you want more than one view device to be installed by your driver, you should override GXDefaultPrinter. That's what the ImageWriter driver is doing.

Q *What is the correct way to recognize PowerTalk letters? Is it by checking for the file type 'lttr' or by looking at Finder flag bit 0x0200? Should I be saving letters as type 'lttr' or as my own file type, and should I be setting the bit?*

A The correct way of checking whether a particular document is a letter is to check the isLetter bit of the Finder info field. It's bit 9, mask 0x0200, and occupies the space formerly used by the "changed" bit.

Checking for file type 'lttr' doesn't work, since letters may have whatever file type the sending application desires. In fact, an application may support multiple letter types, and may have a different file type and icon for each type.

This allows additional flexibility, since an application may have some idea what content the letter contains by checking its type, even before opening the letter.

Note that when your application saves a letter to disk, nothing special needs to be done with either the Finder flags or the file type to ensure that the Finder notices that the file created is a letter. SMPBeginSave and SMPEndSave will automatically set the isLetter bit for you, and the Finder will realize from this bit that your file is indeed a letter.

One final piece of information: Since letters may have a file type other than 'lttr', an additional hook was added to the Finder drag and drop mechanism to allow applications to have any letter drop-opened on them, regardless of the letter's file type. If you add a BNDL/FREF pair with the type 'ltr*' to your application, the Finder will highlight your icon whenever any letter is dragged over it. This mechanism works in the same way that '****' does for the universal drag and drop wildcard.

Q *Can we use a different A5 world with QuickTime? Our plug-in architecture uses A5 for global access, but we allow the A5 world to move. QuickTime doesn't seem to appreciate this and doesn't think that EnterMovies has been called after the A5 world moves. We currently work around this by locking down our A5 world but would rather not. Is locking down the A5 world even good enough?*

A You can use a different A5 world with QuickTime. QuickTime allocates a new set of state variables for each A5 world that's active when EnterMovies is called. However, since QuickTime uses A5 to identify each QuickTime client, if you move your plug-in's A5 world QuickTime will no longer recognize that you've called EnterMovies for that client. So you can use a different A5 world, but you'll have to lock it down.

Q *We're trying to display a QuickTime movie in a frame that can be panned, cropped, and overlaid by other objects. The movie controller doesn't seem to understand that the badge may lie entirely outside the frame. Is there some way to tell the movie controller where to place the badge?*

A Unfortunately, QuickTime isn't flexible about this. The code that positions the badge calculates it from the bounding box of the movie region, and insets it six pixels from the left and bottom. There's no sane way to work around this, other than not to use the standard badge, but instead use your own badge and perform your own badge tracking.

Q *I've noticed some interesting behavior using the standard compression dialog and was wondering if someone could explain it to me. I'm trying to provide session-wide* *defaults for compressing sequences of images. If I don't prime the dialog by doing an SCRequestSequenceSettings, then when I do an SCCompressSequenceBegin the dialog is displayed. Is there any way to prevent this, and to use some set of defaults (without using an image to derive the defaults)?*

A The compression dialog components allow you to get settings with the SCGetInfo call, and to set them with SCSetInfo. The first time, you should display the dialog with SCRequestSequenceSettings, and then use SCGetInfo to retrieve the settings. After that, you can apply the same parameters before starting a compression sequence by using SCSetInfo. If you provide settings before calling SCCompressSequenceBegin, the dialog won't be displayed; otherwise it will be. See *Inside Macintosh: QuickTime Components*, page 3-8 and pages 3-15 through 3-25, for details about the format of the settings.

Also, as you may know, you can generate default parameters that also avoid the dialog by using the SCDefaultPixMapSettings, SCDefaultPictHandleSettings, and SCDefaultPictFileSettings routines. But these do require an image. This way you can avoid displaying the dialog for the first sequence, and still generate valid settings. See *Inside Macintosh: QuickTime Components*, pages 3-26 through 3-28, for more information about these routines.

Q *When a user pastes a movie into a movie-controller movie, the added movie is inserted in the top left corner of the movie. Is there a way for the user to choose where the movie is pasted, and if not, how can I give the movie controller or Movie Toolbox an offset to use rather than have the editing operations use the top left corner?*

A When you paste a movie into a movie-controller movie, the movie controller is simply calling PasteMovieSelection to insert the source movie. All the characteristics of the movie are inserted, and therefore the movie is inserted in the top left corner of the movie. There's no easy way to specify an offset directly to the movie controller. If you want to change the offset of the pasted movie, you'll have to modify the movie yourself after the paste using Movie Toolbox commands. Once you're done changing the movie, be sure to call MCMovieChanged so that the movie controller updates correctly.

The actual modification is simple: call GetTrackMatrix, add your offset to the matrix, and call SetTrackMatrix. The difficulty is in determining which tracks to modify, since the paste may either create a new track or use an existing one. We recommend doing this by gathering all track indexes before the paste, and then comparing with the track indexes after the paste. Since most movies these days have just a few tracks, this shouldn't require much overhead. (But be warned: some movies *do* have a lot of tracks!) To get the track information, you can call GetMovieTrackCount and GetMovieIndTrack.

One last idea: If you don't mind changing the source movie, an alternative is to simply offset the source movie before the paste.

Q *Has Apple defined a codec type for a codec that does CCITT Group IV FAX compression/decompression? If not, how does one go about registering a brand new codec type with Apple?*

A We're not aware of any standard codec type for CCITT Group IV FAX. Apple doesn't have any mechanism in place for registering codec types. We suggest that developers use the creator code for their applications, since creator codes are unique, as long as they register them with us. We hope to have a better solution than this sometime in the future.

Q *When I called Gestalt with the gestaltKeyboardType selector and my new Apple Adjustable Keyboard using System 7.1, I expected Gestalt to return a response that wasn't in GestaltEqu.h, but instead it returned an error of -5550, meaning "Couldn't obtain response." Is there something more I need to do? Should I expect Gestalt to return an error when I get new hardware?*

A Gestalt is in error in not being able to identify the adjustable keyboard. But the real problem (which the adjustable keyboard exacerbates) is that the gestaltKeyboardType selector doesn't really do what you want it to do. What you want Gestalt to do is enumerate the features of a system; however, gestaltKeyboardType tells you the type of keyboard most recently touched, which is of limited utility. In the past, it's been very rare to have more than one keyboard on a system, so the gestaltKeyboardType selector could give you a unique answer. But the adjustable keyboard is two separate keyboards — one for the keyboard and one for the numeric keypad — so gestaltKeyboardType can't give you a unique answer. Because of these problems, we consider this selector to be obsolete.

What you need is a way to list all the keyboard devices on a system. You can use the ADB Manager's GetADBInfo call to get the information for all the ADB keyboards connected. You'll see two keyboards connected if the whole adjustable setup is installed. The first will be at address $02 (all keyboards have an original address of $02); the second will be at an address between $08 and $0F, due to ADB remapping. Your first task would be to determine which ADB devices are keyboards. You can do this by using CountADBs to tell you how many devices there are, and then having a loop that calls GetIndADB for each device. For each device, you would test the original address field; if it's $02, you have a keyboard of some type. You can then use the handler ID of the device to identify the keyboard. The names and device handler IDs of the three main adjustable keyboards and the adjustable keypad are as follows:

U.S. standard adjustable keyboard	$10
ISO standard adjustable keyboard	$11
Japan adjustable keyboard	$12
Adjustable keyboard's keypad	$0E

Note that the keypad will usually be the one remapped and there's no way to guarantee that it will be remapped to a specific address. That's why you have to use CountADBs and GetIndADB to get the information.

Q *Someone told me recently that the glue on postage stamps is made from horse hooves and other mammalian unmentionables. Is this true? I'm a strict vegetarian.*

A The glue used on postage stamps is not made from hooves or any other animal products. Although we're not sure exactly what it *is* made of, we've been assured by the United States Post Office that the glue they use is vegan *and* kosher. (International readers should check with their local postal services.)

Q *Is there any reliable way of extracting the scaling information from a page setup record? We want to display a summary of the page setup in terms of paper size and scaling (for example, 8.5" by 11", 50%) instead of virtual page size (such as 17" by 22"). Can we assume that the editText item with a numerical value is the scaling field?*

A There is no reliable way to extract the scaling information. You certainly can't assume an editText item in a dialog is the scaling field — it could be anything. Don't assume anything about the items or their order, because printer driver developers are free to do whatever they feel like, and most of them have. The ImageWriter allows only 50% reduction, for example, so it's a checkbox, not a numerical field. In some drivers such options may even be in the job dialog for "user convenience."

We've tried for a long time to come up with a reliable way to do this, and there just isn't one. Drivers can and do store scaling values, if they support them, anywhere they like in the print record. You can't compare the page rectangle to that returned by PrintDefault because the user may have picked a paper size other than the default. (You wouldn't want the user to choose legal paper and see 8.5" by 11", scaled 100% horizontally, 122% vertically!) This mechanism was designed to be transparent so that applications wouldn't have to do anything, but it was designed *too* transparently: applications can't tell what the scaling is even if they want to. Note that the new printing architecture of QuickDraw GX actually lets you find out things like this.

Q *I'm trying to draw text to a rectangle and am using TextBox, which works fine. What I want to do is determine the vertical coordinates of the bottom of the text in the text* *box. I want to draw a line right under the bottom of the text, and the text is arbitrary in length. One time it may be a single word, other times an entire sentence that wraps in the rectangle. Do you know any way to calculate what I need?*

A Unfortunately, you can't do it. TextBox doesn't leave the port in any state that can be counted on. However, all is not lost: TextBox is a relatively simple operation and can be easily duplicated with a few lines of code. If you duplicate TextBox, you can access the TextEdit record before it's destroyed. With access to the TextEdit record you can easily measure the text drawn. Try the code below. Also see "The TextBox You've Always Wanted" in *develop* Issue 9 for a very flexible, fast replacement to TextBox.

```
void MyTextBox(Ptr text, long length, Rect *box, short just)
{
// Replacement for TextBox that will draw a line under the last
// line of text.
   TEHandle te;
   Rect     UnderRect;

   te = TEStyleNew(box, box);
   TESetJust(just, te);
   TESetText(text, length, te);
   TEUpdate(box, te);

   UnderRect = (*box);
   UnderRect.top = TEGetHeight(32767, 0, te) + UnderRect.top;
   UnderRect.bottom = UnderRect.top + 1;
   TEDispose(te);
   FrameRect(&UnderRect);
}
```

Q *Is there any way for an application to mount an AppleShare volume in System 6? In System 7, the PBVolumeMount function allows you to do this if you have the right information. Is this call or a similar one supported in System 6?*

A The PBVolumeMount, PBGetVolMountInfoSize, and PBGetVolMountInfo calls are all supported in System 6 *if* the AppleShare 3.0 or later Chooser extension has been installed. These versions of the Chooser extension have been tested back to System 6.0.4, and work fine.

Q *Our product needs to access the DCD signal from the modem port to function correctly when talking to an external modem. Most documentation doesn't say anything about the DCD signal being available. However, we have a hardware diagram that shows that pin 7 on the mini-8 connector is connected to give a DCD signal on later*

Macintosh models. In most documentation pin 7 is shown as not connected. Is pin 7 connected to anything, so that I can wire it to the DCD signal from the modem? If so, how can I access the status of pin 7 to see when DCD is asserted or negated? If I can do this, what models of Macintosh will it work on, and will it stay like that in future releases?

A There is ongoing confusion regarding the Macintosh serial ports and how to wire them for serial operation. Macintosh serial ports are RS-422. By shorting the positive receive pin to ground, you create an RS-423 port, which is an RS-232 emulation. The table below shows a pin-wiring schematic from the Macintosh DIN-8 to DB-25 RS-232 serial cable, which supports hardware handshake and DCD transmission (arrows show the direction of signal transmission):

DIN-8		**DB-25**
HSKo	1 —> 4, 20	RTS, DTR
HSKi	2 <— 5	CTS
TxD–	3 —> 2	TxD
GnD, RxD+	4, 8 — 7	GnD
RxD–	5 <— 3	RxD
TxD+	6	
GPi	7 <— 8	DCD
Shield	1 — 1	Shield

Notice that we connect the DCD signal to the GPi input pin. As you know, the GPi trace isn't implemented on all Macintosh CPUs. Additionally, there is no serial driver call implemented to obtain the current state of the GPi pin. Rather than have a complete list of which CPUs it's connected on, you can fairly easily make your application dynamically find this out on a particular CPU at run time. So you can tell the user whether DCD from a modem can be detected on a particular installation, with no need for a priori knowledge. See the Macintosh Technical Note "Serial GPi (General-Purpose Input)" (Devices 16) for a description of using GPi and Gestalt.

Q *I use the OpenPicture, draw, ClosePicture sequence to create a picture handle. Since the handle can be quite large, and since I dispose of it fairly quickly, it would make sense to allocate it in temporary memory. But I haven't found any reasonable way to do that. Any suggestions? Both these situations arise because my application runs in a fairly small (800K) partition. I do this so that other applications have adequate space to work with it, since one of my main functions is to interact with other applications using Apple events. However, I occasionally need more memory for a few seconds at a time.*

A There are two ways to cause OpenPicture to use temporary memory. A simple way to do it is to allocate a block of temporary memory, then create a new heap zone in that block and make it the current zone just before you call OpenPicture. This will cause subsequent memory allocations to happen in your temporary block, and will work fine. See *Inside Macintosh: Memory* for more information about creating heap zones.

Another way is to replace the putPicProc, as is commonly done when spooling a picture to disk, and instead spool it to temporary memory. If you're not familiar with picture spooling, see *Inside Macintosh* Volume V, page 89, which has code for spooling a picture to disk as it's created. This is the same technique as documented there, only it spools the picture to temporary memory instead.

You create a handle in temporary memory the size of a picture and fill in its size and picFrame fields so that it looks like a normal picture handle. In your putPicProc you copy the data in, continually resizing the handle if necessary to fit the data. After you call ClosePicture, remove your putPicProc; then you can use the temporary handle just like a normal picture.

The advantage of this method over the first one is that you can make the picture as large as temporary memory will let you, and you end up using just enough memory. The first technique, while technically easier to implement, limits you to the size of the heap you initially create, and you also may use a lot more temporary memory than you need. If you're able to come up with a good guess of how large your pictures are going to be, use the first technique; if not, use the second one.

Q *I think I may have found a bug in the System 7 TrueType rasterizer code: under some clipping conditions it may crash/freeze the system inside the DrawText call. I'm really not doing anything out of the ordinary; the key to making it die apparently lies in the placement of an obstructing window that just barely overlaps the text being drawn. Investigating the problem in MacsBug reveals that somewhere deep in the TrueType rasterizer some code is trashing the A3 register and not restoring it properly.*

A This is a bug in QuickDraw's text-drawing routine, introduced when support for TrueType was added: QuickDraw (under certain circumstances, like foreground not black or background not white) allocates an off-screen buffer for the text on the stack. If there isn't enough stack space, the string is cut in pieces, and the code reentered for each piece separately. (If that still doesn't help, no text is drawn at all.) The bug is that A3 is assumed to hold the port throughout the text-drawing code, but when we added support for TrueType, we reused A3, forgetting the previous convention for the case where the string had to be subdivided. This probably won't be fixed before the rewritten QuickDraw is released in PowerPC processor–based machines.

The workaround is to preflight available stack space before issuing text-drawing calls for long strings and large point sizes, and draw the string in pieces yourself if necessary. You may also want to allocate more stack space at startup via

```
SetApplLimit(GetApplLimit() - something);
```

with *something* empirically determined. For a big application with a large minimal partition size, 16K or 32K more or less in the heap shouldn't matter; but it's a big win for QuickDraw's reliability, and generally a good recommendation.

Q *I would like to be able to determine the clock speed of any CPU. I know that I could write a routine and calculate the time it took to execute the code, but can you tell me a more direct way?*

A You can use the GetCPUSpeed routine to check the current clock speed of the PowerBooks (after checking gestaltPowerMgrAttr to make sure that the Power Manager is present). This routine is documented in the Power Manager chapter of *Inside Macintosh* Volume VI (page 31-20).

Unfortunately, there's no way to check the clock speed of other Macintosh models. The only thing you might be able to use as a crude way of comparing relative speeds is the low-memory global, TimeDBRA ($0D00). TimeDBRA gives you the number of iterations of DBRA per millisecond, which can help in a relative speed comparison.

Q *A wonderful thing about MacsBug is that you can save logs, and our testers use it to log crashes of our programs under development. Unfortunately, some of our more lazy testers just send a log without a description of how they made the crash occur. I was wondering if there's a way a user can enter a comment in MacsBug that will get saved in a log but won't be interpreted by MacsBug. This way, users could put the whole bug report in one nice consolidated log, without going and editing it later in a text editor.*

A You can enter text into your MacsBug log by using the printf dcmd in MacsBug. It's similar to the C printf function. If your testers want to add anything to the log file, they can simply type

```
printf "interesting comment that I wanted to make"
```

For more on the printf dcmd, see the MacsBug 6.2 reference manual.

Q *I have problems linking my QuickDraw GX printing extension when I include MPW's StdClib.o, evidently because I call sprintf. I was told this is because of global variables in* *the StdClib.o library. Does MPW have a parallel library for use with standalone code — something like THINK C's ANSI-A4 library?*

A The problem is that two of the routines in StdCLib.o use global data. You can get around this problem by redeclaring the problem routines in your code. First #include <StdIO.h> in one of your source files or included header files. Add the following to your source:

```
size_t fwrite(const void *, size_t, size_t, FILE *) { return 0; }
int _flsbuf(unsigned char, FILE *) { return 0; }
```

These lines redefine fwrite and _flsbuf, the offending routines, so that they don't use global data (or do anything, for that matter, except return 0). Make sure that the object file containing the above routines appears in your Link instruction *before* StdCLib.o, since only the first declaration will be used. Of course, this means that you can't use the fwrite and _flsbuf routines, but it's unlikely you'll want to in a printing extension anyway. Also, sprintf never calls them. The linker will generate "duplicate symbol definition" warnings for the two routines, of course, unless you specify **-msg nodup** in your Link instruction. In addition, don't forget to link with RunTime.o.

The sprintf routine is great for debugging, since you can easily get formatted debugger output, as follows:

```
OSErr    err;
Str255   pStr;

// Do stuff that might cause an error
. . .

if (err) {
   pStr[0] = sprintf(&pStr[1], "Error %d", err);
   DebugStr(pStr);
}
```

Also, check out the dprintf command in QuickDraw GX's GXExceptions.h file; it's similar to this.

Q *What is the correct term for the little depression above the center of my upper lip? In high school we called it a "curvicle," but I'm pretty sure we made that up.*

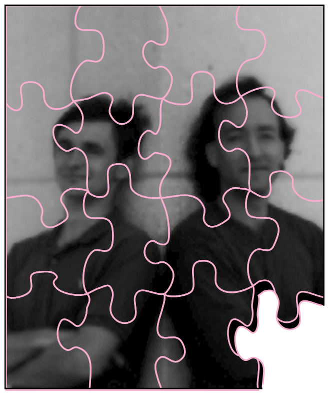
A The correct term is "philtrum." Biologists debate heatedly about its real purpose, but there's no doubt that it makes a good place to rest the tip of your index finger during periods of deep thought.

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

WHEN MAPS GO BAD

See if you can solve this programming puzzle, presented in the form of a dialog between Konstantin Othmer (KON) and Bruce Leak (BAL). The dialog gives clues to help you. Keep guessing until you're done; your score is the number to the left of the clue that gave you the correct answer. These problems are supposed to be tough. If you don't get a high score, at least you'll learn interesting Macintosh trivia.


KONSTANTIN OTHMER AND BRUCE LEAK

KON: I have a Macintosh IIfx at home with a 13-inch color monitor and an old Kong.

BAL You mean the black-and-white two-page display? That thing is out of production. Programmers were the only ones who bought it, and we already sold one to every Macintosh programmer. I've got one too.

KON Mine's probably older, though. Fung gave me one of the prototype units that were blocking the entrance to his office about five years ago.

BAL Does it have the stylish metal cheese grater cage around it or did you actually get a plastic case?

KON I scored one with plastic, though it's yellow now from sitting in the sun. Anyway, now that the PowerPC workload is finally winding down, I've actually spent some time at home. I started playing around with Darryl's mapping program on my fx.

BAL MacAtlas?

KON Yeah. It's pretty cool. The problem is I can't open files.

BAL Command-O open? What happens?

KON Nothing. No Standard File, no dancing bears — it's as if I never gave the Open command.

BAL I've never had a problem with that program. Does the menu get highlighted?

KONSTANTIN OTHMER AND BRUCE LEAK regularly change their AppleLink addresses through various front companies to help defray their tax losses from dealings in the stock market. They're currently trying to corner the market on masked ROMs, particularly 680x0 ROMs targeted at PowerPC processor–based machines. They were last seen looking for a '57 Corvette in good condition and are willing to swap debugging services in trade.•

KON Yeah. Every once in a while you'll see it flash. This problem happens systemwide; I can't even open files in ResEdit.

BAL Can you drag documents onto the application icon in the Finder to open them?

KON Yeah. Everything not related to Standard File seems to work fine. I can edit files, play movies — even file sharing works.

BAL What version of the system are you running? Does printing work?

100 KON I'm running System 7.1. I can't save and then print as recommended by our print shop because there's a problem with Standard File, but other than that it works fine. Why?

BAL This one's easy, KON. Something is locked down in your MultiFinder heap, so your system heap can't grow. Standard File can't bring in all its resources, so it punts. Are you running that skanky StockItToMe server that Timo and Rubinowitz wrote?

90 KON This happens just after boot and I've got mounds of RAM now that I've removed DAL. I do a heap dump (**hd**) of the MultiFinder heap and there are no locked blocks at the bottom, so the system heap can grow all it wants. Furthermore, the heap total (**ht**) command shows that there's plenty of room in the system heap.

BAL Clearly this bug is limited to your home machine. I figure you have some slimy KON code on there that's only half debugged — maybe a beta version of WonderPrint. Set an ATB on Standard File, 'PACK' 3, and then try to open a document.

80 KON I hit the A-trap, but when I trace over it it just returns without putting up the Standard File dialog.

BAL Is the package getting loaded?

70 KON Yeah.

BAL Since I don't have the source to Standard File handy, I'll use the log command and then the conditional step command to get a trace of all the instructions executed inside the package. I figure it should be fairly easy to spot some error condition where the code decides to bail. Since the _Pack3 instruction is two bytes, if the call to it is at the current PC address, I use the MacsBug commands

```
log MyStdFileLog
s pc=pc+2
```

to step, logging each instruction to a file and then stopping as soon as the trap is exited.

60 KON Wow, that conditional stepping is pretty cool. I didn’t know MacsBug was that sophisticated. You could have just put a breakpoint on the other side of the _Pack3 and stepped a zillion instructions.

Anyway, the trace isn’t prohibitively long. Standard File preloads all the resources it needs to display the dialog, checking each one to see if it was loaded properly. When it tries to load LDEF -4000, it fails and bails.

BAL All the resources it needs must come from the System file. Check to see if LDEF -4000 exists in the system.

55 KON According to ResEdit, the LDEF isn’t present in the System file.

BAL Compare the LDEFs in your system with a fresh installation.

50 KON The LDEFs are all the same except one, which has a different resource ID.

BAL Well, it sounds like you’ve got a trashed System file. Copy and paste the bad LDEF into your system and reboot.

45 KON Everything works fine now, but if I drag-install a new system, Standard File fails. Fixing the LDEF is addressing the symptom, not the problem.

BAL You drag the good System file over and reboot, and Standard File is broken again?

KON Yep.

BAL Check the LDEF with ResEdit.

40 KON It’s renumbered again.

BAL So something is trashing the resource map during boot. What if I boot with the Shift key held down to disable extension loading?

35 KON According to ResEdit, the resource map is still trashed.

BAL Try dumping resources of type 'LDEF' with the command **rd LDEF** at interesting times during the boot process.

30 KON If you hold down the Control key to enter MacsBug just after it’s installed, the LDEF resources are fine. If you check it again when you’re in the Finder, the LDEF is bad.

BAL So I use

```
atb hopenresfile ‘;dm @(sp+2);rd LDEF;g
```

to watch extensions load during the boot process. The command breaks as each extension is loaded and displays the name of the extension and the current state of all LDEF resources. When I see the LDEF go bad, I've gone a long way toward finding the offending code.

KON During the 7.1 boot process, the system patches are loaded from disk and installed. One of those patches opens all the font files in that crazy font folder. After all the patches are installed, the system extensions are loaded, starting with the 8•24 GC card. It's loaded first so that QuickDraw is in a well-defined state before other system extensions go and patch out the world; once third parties start getting in there, it's a free-for-all.

BAL Obviously, all the system patches need to be installed before extensions are loaded so that the extensions can take advantage of improvements provided by the patches. Haven't you ever wondered why booting takes so long?

25 KON While the font files are being opened, the resource map is fine. But just before the first extension is loaded, the resource map is bad.

BAL We need to narrow down where the map is going bad. I'll look through the system map for the ID that's getting stepped on. First I get the size of the resource map handle by using

```
wh @@sysmaphndl
```

Then I can find the resource ID with

```
f @@sysmaphndl size f060
```

since F060 is hex for -4000, the resource ID that's getting smashed.

20 KON Good try, but you find seven occurrences.

BAL So I continue until the map goes bad, and check for -4000 again. I'll find the one that's changed, reboot, and step-spy on that address. Ta-da!

15 KON Sorry, but the resource map is a handle and it has moved on you. You can identify the one that's changed, but you can't step-spy on it since it's moving around all the time.

BAL Fine. I'll reboot in 24-bit mode, lock the handle down by setting the high bit of the resource map handle's master pointer, and step-spy on the address. Over.

10 KON Now Standard File works fine, but file sharing is starting to act flaky.

BAL OK. So I'm going to have to brute force it. I'll break when the last font loads. I'll log the MacsBug output to a file with **log myLDEFFile** and then **atb ';rd LDEF;g** to break on all traps and dump all the

SCORING

80–100	This is a special category reserved for Waldemar and Jasik.
50–70	Weekends spent at Weird Stuff Warehouse or Van Brink's garage are starting to pay off.
25–45	You're spending too much time playing with that 3DO interactive multiplayer machine at your local electronics store.
5–20	You're spending too much time at the gym.•

LDEF resources. I'll also do an **atc hopenresfile** and then an **atb hopenresfile** to clear the **rd LDEF;g** when HOpenResFile is hit. This way we'll fall into MacsBug when the first INIT is opened. Then I'll let it rip and get some lunch.

5 KON Pretty snazzy there, BAL. Before the map goes bad you see a bunch of Slot Manager calls and control calls. The last trap that gets called before the map goes bad is a control call to the driver which seems to originate from within InitGDevice.

BAL Hmmm. We're at secondary INIT time. The driver gets called again since all the system patches are now loaded. We added this because monitors boot back to the same configuration they were in when the machine was shut down. With 32-Bit QuickDraw, the machine could have been shut down in 32-bit video mode, so the driver needs to be called a second time after the system patches (where 32-Bit QuickDraw lives) are loaded.

KON Yeah, the call that's killing the map is a call to the driver to set the bit depth. At secondary INIT time the 'scrn' resource is read to find out what the last bit depth was, and each device is called to set the depth up properly.

BAL The call must be going to the bogus Kong video card Fung gave you. What's the ROM version on that thing?

KON It's a beta ROM. When I asked Mike Puckett about it he said there was some nasty "*fung*us" in the code, and that it was fixed before final. He gave me a new video ROM, and now everything works great. Real-world users would never experience this, of course, since the problem was corrected long before the card went out.

BAL You and Van Brink should share packrat stories. You never know, some of that old equipment might become collector's items some day.

KON This problem began when I started using MacAtlas because I switched my Kong into grayscale mode. My system is a lot more stable now that those spurious writes have been fixed. Before I tracked this bug down, I was having intermittent problems.

BAL Nasty.

KON Yeah.

Thanks to Peter Hoddie, Brian McGhie, and Mike Puckett for reviewing this column.•

HISTORY OF THE DOGCOW

PART 1

MARK ("THE RED") HARLAN

I'm going to tell you a few things that have *never* been put in print before about the dogcow. If you don't know what or who the dogcow is, or you don't care for Apple cultural minutiae, you should just flip past this column.

This is only part 1 of the story, to be followed by more in a future issue of *develop*. We didn't want to hit you with it all at once, for fear of what the shock (or the boredom) might do.

HOW IT ALL BEGAN

The dogcow was originally a character in the Cairo font that used to ship with the Macintosh; it was designed by Susan Kare. I had always been interested in this critter ever since I first saw it in the LaserWriter Page Setup Options dialog, sometime during my stint in Apple's Developer Technical Support (DTS) group in 1987. To me it showed perfection in human interface design. With one picture it was very easy to explain concepts like an inverted image or larger print area that otherwise would be nearly impossible to communicate.

Interest became an obsession when one day I was talking to Scott ("Zz") Zimmerman about the dialog and suddenly thought, "Just what is that animal supposed to be, anyway?" Since Zz was the Printing Guy in DTS (now in the Newton group), and my favorite pastime was to bother him endlessly anyway, I started pressing him on whether the animal was a dog or a cow.

In an act of desperation he said, "It's both, OK? It's called a 'dogcow.' Now will you get out of my office?" The date was October 15, 1987, and I consider this to be the first use of the term. It should be noted that since then a few people (including Ginger herself) have told me that actually the phrase was coined by Ginger Jernigan (ex-DTS, now ROM software) at a meeting of Apple's Print Shop sometime shortly before that, which very well could be the case. Nevertheless it was Zz who pressed it into common usage, and he certainly was the first person I ever heard use the term.

Zz's ploy to get me out of his office was futile, however, because then I stood around and postulated that the dogcow's genes would have a radical effect on its behavior, and it must not bark or moo, but rather utter a combination like "Boo-woo!" or "Moof!"

We both thought it was funny enough that we decided to press it into everyday usage, and I started circulating the dogcow with "Moof!" on internal memos. The idea caught on, and at the 1988 Worldwide Developers Conference we gave away dogcow buttons in the debugging lab. Louella Pizzuti (ex-DTS, ex-*develop* editor, now citizen of the world) came up with the great idea of making the background Mountain Dew green. Response to the buttons was huge, and no one was smiling more than the DTS folks when John Sculley wore one for his keynote speech. It was a major-league coup.

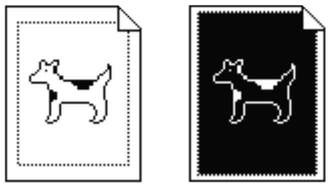
THE ORIGIN OF TECH NOTE #31

Then things started to spin out of control. Various groups internally started picking up the dogcow logo and doing things that didn't seem, well, DTS-like. The final straw was when the dogcow pin appeared in a Microsoft advertisement. Mark Johnson (ex-DTS, now in Apple Europe) approached me and suggested that we throw down the gauntlet and write a Technical Note on the subject. I balked out of nothing more than sheer laziness.

Some time passed and we were getting ready to go with the April 1989 batch of Tech Notes when Mark approached me again, saying that he thought having an

MARK ("THE RED") HARLAN started life in Rawlins, Wyoming, and has led about exactly the kind of life you'd expect as a result. He spends most of his time at Apple finding employees who were hired by Steve Jobs and asking them, "So how does it feel knowing that the way you changed the world is by putting Windows on all PCs?"•

Our friend in the LaserWriter Page Setup Options dialog, normal and with Invert Image and Larger Print Area:



April Fool's edition describing the dogcow would be perfect. I said yes but then stalled and stalled, missing two deadlines, and I thought the Tech Note wasn't going to happen.

Mark marched in my office one day in March of 1989 at 11:30 A.M. announcing that Tech Notes were shipping at noon and implied that my manliness was in question if I didn't get that Note in the batch. My macho instincts just couldn't allow that to happen, so Tech Note #31, "The Dogcow," was written in literally 40 minutes in one pass. I'd been thinking about it for quite some time, so I knew pretty much how it would go; I just sat down and typed it out. Given more time I definitely would have churned out something a bit more polished, and part of its quirkiness, I'm sure, is due to the time pressure I was under.

One thing was certain: it had to be something original in concept. I've always had a deep disdain for people who rip off comedic stuff. You know, the same people who used to have to tell all their jokes with an English accent because of Monty Python are now those who say "*Not!*" behind phrases. Once is funny, but after a while it gets really old. I definitely wanted it out of the mainstream.

For numbering I wanted to use *e*, but Mark pointed out that there had been confusion early on in the Tech Note numbering scheme and that a few numbers had been left out for various reasons. He showed me some conversations from the net that went on and on about Tech Note #31 and people's guesses as to why it was missing. (People were really, really out there with their guessing; anyone who's a believer in conspiracy theories would have enjoyed this blatant gibberish.) The number 31 had the right feel; it would blend into the regular batch better than *e*, and I've always had a soft spot for prime numbers, so we picked it.

Sports Illustrated had run a great fake story about a Zen baseball pitcher sometime earlier and we borrowed the idea of having the words "April Fool's" spelled out within the article from them — in our case using the first letter of every line of the poem at the closing. No one has ever mentioned this to me, so few people must have caught it.

There's a picture of the wrong way to draw the dogcow that several people thought was a scanned image of Zz. Actually, completely independently of the Tech Note, I'd been using a program called Mac-a-Mug, designed to make mug shots, and came across a set of hair that looked frighteningly like Zz's. After fiddling around with the program a bit I was able to come up with a good rendition of Zz's head, and I shoved it into the Tech Note without his ever knowing about it. The expression (and color) of his face when he learned about the picture is a memory I'll always cherish.

The Note also contains the expression "Aanal, Enacku Naiimadu, Kaanali!" People came up with very unusual anagrams or unusual explanations for what it meant, the best being that it was an obscure reference to a clip of *The Day the Earth Stood Still* that had been cut from the film. But the truth is that it's phoneticized Tamil that was supplied by Sriram Subramanian (Networking Guy, ex-DTS, ex-Taligent, now in Apple Japan) meaning "But I can't see the dogcow!"

Ironically, there's also a mistake in that the "correct" way to draw the dogcow is actually wrong. We ended up being so pressured for time in getting the Note out the door that we just jammed it into a weird PostScript file that ended up mutating the shape. Shortly after the release of that Note, Chris Derossi (ex-DTS, now at General Magic) convinced me that a better solution was to have the correct way to draw the dogcow be pixelated, to avoid these idiosyncracies in the future — which is what's now done.

NEXT TIME

There will be more history of the dogcow in a future issue of *develop*. Have you ever wondered if you have the entire set of dogcow pins? Is that dogcow T-shirt of your cubemate's bootlegged? Is there any way things can get more meaningless? Some, but not all, of these questions will be answered the next time we have a little extra space to fill.

Tech Note #31 is not on this issue's CD and hasn't been on the CD for quite some time now; it's no longer available. It used to be hidden in the Technical Notes Stack on the early versions of *develop*'s CD. It appeared on paper only once, as part of the monthly mailing to Apple Partners and Associates in April of 1989. The continued secrecy has a little bit to do with history and a lot to do with tradition. For more on the distribution of the Tech Note, stay tuned for History of the Dogcow: Part 2.•

Thanks to Gary Robinson, whose letter asking for the story of the dogcow inspired this column.•

INDEX

For a cumulative index to all issues of *develop*, see this issue's CD.•