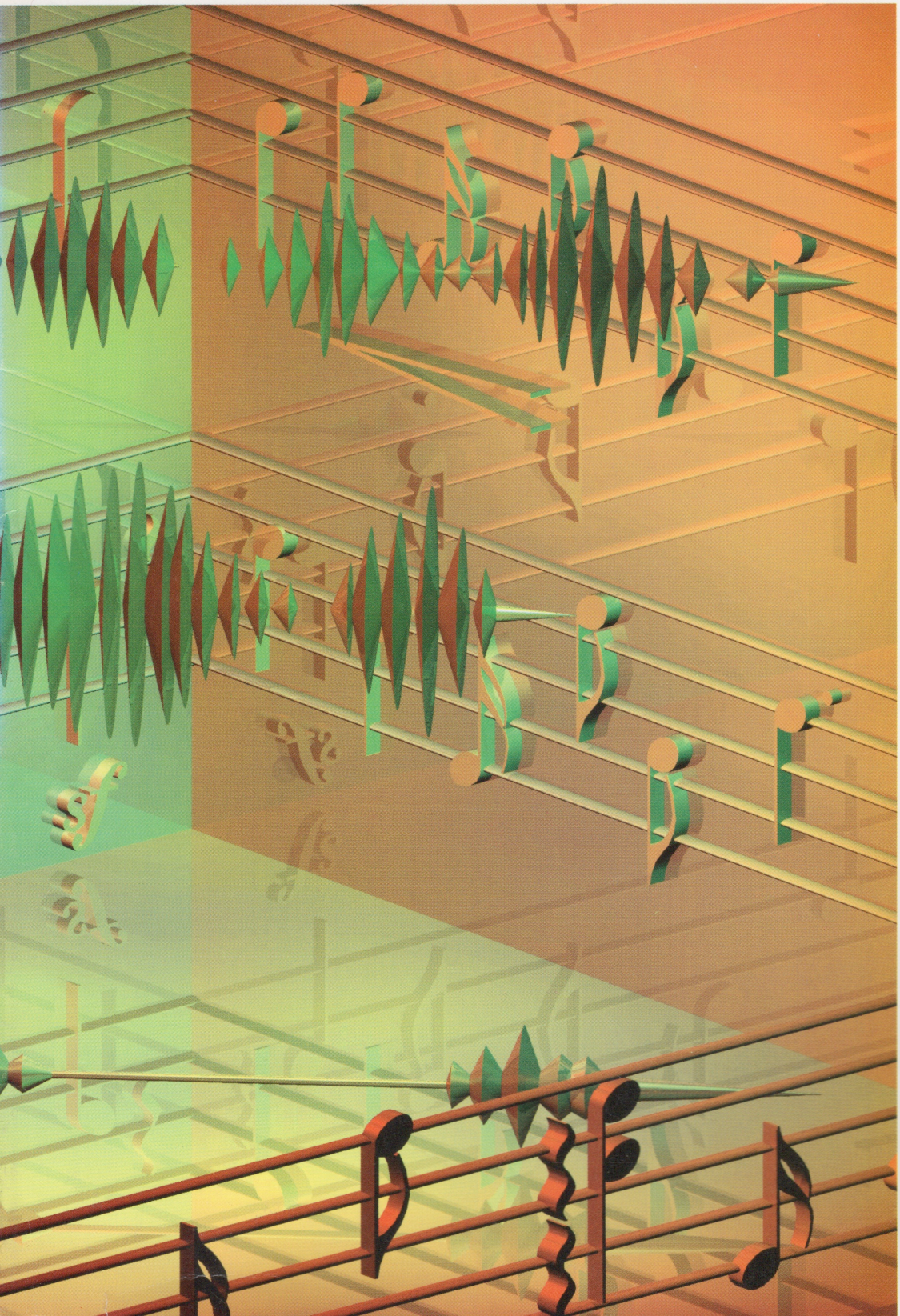
develop

The Apple Technical Journal



Issue 11 August 1992

EDITORIAL STAFF

Editor-in-Cheek *Caroline Rose*

Technical Buckstopper *Dave Johnson*

Our Boss *Greg Joswiak*

His Boss *David Krathwohl*

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

Managing Editor *Monica Meffert*

Assistant Managing Editor *Ana Wilczynski*

Contributing Editors *Lorraine Anderson, Toni Haskell, Judy Helfand, Rebecca Pepper, Rilla Reynolds*

Indexer *Ira Kleinberg*

ART & PRODUCTION

Production Manager *Hartley Lesser*

Art Director *Diane Wilcox*

Technical Illustration *Nurit Arbel, John Ryan*

Formatting *Forbes Mill Press*

Printing *Wolfer Printing Company, Inc.*

Film Preparation *Aptos Post, Inc.*

Production *PrePress Assembly*

Photography *Sharon Beals, Lisa Jongewaard*

Online Production *Cassi Carpenter*

Cleo Huggins of Rucker Huggins used Ray Dream Designer 2.0 to orchestrate this 3-D score of sound and music. The music symbols are characters from Sonata, a noteworthy typeface that Cleo designed at Adobe Systems.

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

The *Developer CD Series* disc for August 1992 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.

CAROLINE ROSE

Dear Readers,

Time marches on, and with it, inevitably, comes change. I'll be telling you here about some recent changes in the world of Macintosh documentation and *develop*. We think they're changes for the better—but of course you, the developer, are the final judge.

First, the "bible," *Inside Macintosh*, is on its way out, starting with the imminent publication of *New Inside Macintosh*. I know only too well what your problems were with the first three IM volumes, which I slaved over for a good chunk of my life—not enough examples, not enough explanation of how the parts work together as a whole—and (till now) I've resisted the urge to defend myself here by telling you how time- and staff-restricted we were, how the software kept changing out from under us, and numerous other excuses. Volumes IV through VI sparked new complaints, primarily that the information on a specific topic was scattered over several volumes. All these problems have been addressed in *New Inside Macintosh*, which brings everything together in an organized and understandable way that should, once it's published in its entirety, have you happily discarding all your old volumes (except for you sentimentalists who will never part with your old "phone book" edition). The first NIM books are due to appear in bookstores in September, and the last books in the series should be available by May 1993. The electronic versions of these books will show up on the *Developer CD Series* disc as soon as they're ready. References to *Inside Macintosh* in *develop* will continue to point to the original IM volumes until next year when the transition to NIM is complete.

Tech Notes have also undergone a reincarnation, as you'll notice when you look at them on the CD. Our new Tech Note poobah, Neil Day, talks about this at the end of the Letters section, on page 6. The Notes are no longer numbered, but are now organized by subject, similar to the organization of *New Inside Macintosh*. As a result of this change, *develop*'s references to Tech Notes now refer to numbered Notes as things of the past (for example, "formerly #161").

Another change in this issue of *develop* is one we want to be sure you know is an aberration: There's no Print Hints column this time. It turns out that Luke Alexander committed one of the very printing crimes he wrote about in Issue 10, and he couldn't pay the bail. Well, actually, Luke was very busy preparing for his talk at the Worldwide Developers Conference while this issue was being written, and we had to let him off the hook. He promises he'll be back stronger than ever in Issue 12.

CAROLINE ROSE (AppleLink: CROSE) first interviewed at Apple in 1982, when she was shown a Macintosh with balls bouncing all over its screen. Having been raised on computers as card sorters and number crunchers, she thought this was pretty exciting, and signed up to write its technical documentation. Her love affair with the Macintosh suffered a blow when she left in 1986 to join NeXT, but she came back after five years away, and all was forgiven. Caroline recently learned from a bio in an early issue of *develop* that 8/8/88 was considered a very lucky day by the Chinese, and she ponders the significance of having missed that day entirely due to crossing the international date line. She loves to travel, whether in planes, trains, and automobiles, on foot, or simply back and forth in a swimming pool.•

Speaking of the Worldwide Developers Conference, it was wonderful to meet so many of you there, hear your good words about *develop*, answer your questions, and set you straight on a few things. Many developers didn't know, for example, that *develop* accepts articles from outside Apple (though we rigorously review them just like our own) and that they don't have to already be at the polished level of writing you're used to seeing in *develop* (we have editors who help with that). The overwhelming majority of Associates and Partners expressed their displeasure at no longer receiving *develop* in printed form in their monthly mailing (though some of them still hadn't realized this, because it usually takes a long time for *develop* to get to them anyway). Not all of them knew that they could subscribe to get printed *develop* (through APDA, AppleLink DEV.SUBS, or the subscription card in an actual printed issue). We'll keep trying to spread the word; meanwhile, please tell two friends.

On to the trivia . . . Issue 10's editorial asked: What character in *develop*'s body font is upside-down (not just one-time-only, but defined that way)? The answer is "8"—see? If it were a snowman, it would topple over.

Some of you wondered about my answer to this earlier trivia question: What word was used instead of "click" to describe the action of pressing a button on that first mouse? The answer was "bug," and I was asked whether the choice of that word was a joke, or what. I called Doug Engelbart himself to find out, and he said that in those days the cursor was called a "bug," so it became short for "to put the bug somewhere by pressing this button." It had nothing to do with the meaning of "bug" as a problem in a program. As to why "bug" was used to refer to the cursor, he didn't know the history of that. So I guess—unless one of you can shed some light on this—the bug stops there.

Caroline Rose
Editor

SUBSCRIPTION INFORMATION

You can subscribe to *develop* through APDA (see the inside back cover) or by using the subscription card in the back of this issue. Please address all subscription-related inquiries to *develop,* Apple Computer, Inc., P.O. Box 531, Mt. Morris, IL 61054 (or AppleLink DEV.SUBS).•

BACK ISSUES

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

LETTERS

DEVELOP ON FTP.APPLE.COM

I think *develop* is the greatest Macintosh journal around. I've gotten lots of help from the articles and the code on the CD. One thing about the CD: It would be nice if I could ftp the files from apple.com, since I don't have a CD-ROM player.

—Jim Wintermyre

Thanks for the kind words about develop, *and for the idea of putting the files on apple.com (actually, it's now ftp.apple.com). Like Technical Notes, DTS Sample Code, and Snippets,* develop *articles and code are now available via anonymous ftp on ftp.apple.com, thanks to Mark Johnson, manager of Apple's Core Technical Support group, who does this on his own time.*

—Caroline Rose

OUR AUTHORS ARE REAL

I would like to suggest that you publish your authors' e-mail addresses. I wanted to send a note of praise, thanks, and encouragement to Bryan K. ("Beaker") Ressler, author of the excellent article "The TextBox You've Always Wanted" in Issue 9— but I didn't know how to reach him. Even a "Find Address" search on AppleLink turned up nothing. Are you sure this guy's for real?

—James Plamondon

Yes, we're sure. Since receiving your suggestion, we're asking all authors if they'd like to put their e-mail addresses in their bios. Many would prefer not to be contacted directly. And sometimes the authors are in flux and don't have a stable or convenient address for a while. Where no address is provided, letters should be sent to the AppleLink address DEVELOP, and they'll be forwarded.

—Caroline Rose

ASSOCIATES MISS DEVELOP

In Issues 8 and 9 of *develop*, you mentioned that Apple Partners and Associates no longer receive a printed copy of the publication. If I were an Associate, and I plan to become one soon, I would continue to subscribe to the printed version.

First, there's the "curling up in front of a fire" factor you mentioned. There are many places I take *develop* that I couldn't take a Macintosh and CD-ROM drive. It slips easily into a briefcase and can be read on a bus or while waiting in line. While I'm reading *develop*, somebody else can use the Macintosh. Some of the articles require some effort to understand, which is easier while sitting in an overstuffed chair with the article in my hands than while looking at a computer screen. Oh, yeah, put a cat in my lap for good measure.

Second, the aesthetic experience of the printed version would be hard to give up. The beautiful covers are the most obvious part of this, but the care you put into laying out the pages, providing just the right amount of white space, selecting the typefaces to complement each other and make the content easy to read makes reading *develop* a very pleasant experience. Then there's the smell of a newly printed *develop* and the faint "crick" sound of it being opened for the first time.

Third, the printed version has material that would be impossible to view on the 1-bit screen I'm using. How would I know what "Konenna" looks like

IT MAKES OUR DAY WHEN YOU WRITE

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

without a printed copy? How would I know what *you* look like? Even if I could display some of the artwork and photographs, it would take some effort and time, and I might not bother. Turning a page is a very simple thing to do. And the access time is much faster than a CD-ROM could ever be.

Fourth, the "green" factor. A Macintosh uses electricity even when you're just sitting there staring at the screen. I haven't noticed *develop* using any, no matter how long I spend reading it. I believe you use partially recycled paper, and you don't have to worry about my copies being recycled because they'll sit on the bookshelf until I die or get a room in the loony bin. Sort of like *National Geographic*.

develop, in its physical form, is beautiful, informative, sometimes funny, interesting, and occasionally inspiring. I won't give it up when I become an Associate, though for what that costs I darn well shouldn't be asked to. Keep up the good work. I appreciate it!

—Lyle Gunderson

I've been a fan of *develop* since its first issue, and still look forward to each one. I take mouse in hand to share my angst about hardcover versions of *develop*.

I love them.

Hey, we recycle at home. We have a compost heap. We're down to one car. But, for a mag like *develop*, I like the feel of hefting it, scanning through to see what's up, and snooping through the bios looking for the ever present chuckles. It's a magazine I sit on the couch with and, what else, browse. This is not the Technical Notes stack, nor is it *Inside Macintosh*. I'm prepared to look at them through the glass keyhole of my Apple monitor, since they're only mildly amusing, very functional, and I usually need them when I'm in front of it anyway. That's what references are for.

But *develop* is a different beast. *develop* is *Life* magazine for Mac-crazed software craftspeople. It needs to be perused, thoughtfully, where it can be set down and have some latte dripped on it.

I hope the developer mailing resumes the practice of sending *develop* in paper form.

—David Kauffman

I love these letters—and the similar comments that we got at this year's Worldwide Developers Conference. As a result of all this feedback, which has been pouring in since develop *was taken out of the developer mailing, people in high places at Apple no longer believe that this is what most Associates and Partners want. No changes are imminent, but the subject is not dead. For now I can only urge you to pay the $30 to subscribe to* develop *in its printed form.*

—Caroline Rose

NEW SINCE LAST DEVELOP?

Could you prevail on the Developer CD folks to include a "What's new since last develop" folder on CDs that come with *develop*? It would be done like the very nice "What's new on this CD?" folder, but cover three months rather than one.

—John Baxter

The "What's new on this CD?" folder now indicates (separately) what's new for each of

the last three months—in this case, June, July, and August. Thanks for the idea!

—Caroline Rose

FAKIN' IT

develop is a great publication, and the Q & A section is especially useful. In Issue 10 you define the term "fakey" as "riding your snowboard backwards." There are two problems with this. First, the correct spelling is "fakie." The greatest mistake, however, was not attributing the word to its original source: skateboarding. Snowboarding has taken almost all of its trick names from skateboarding, "fakie" included. Keep up the good work, but get the facts straight! ;-)

—Frank Giraffe

We here at develop *are embarrassed and chagrined that such an obvious and important error could have slipped past us, and we apologize profusely for any inconvenience this misrepresentation of fact has caused you, either real or imagined. Thank you for your comments.*

—Dave Johnson

TECH NOTES TAKE A NEW PATH; CHECK IT OUT!

BY NEIL DAY

It's 3:00 A.M. You've just finished polishing that code that you've been slugging it out with for the last several nights. It works beautifully, you're happy and full of energy. Instead of waking up the nearest person for a demo (and risking being murdered), you decide to start work on "just one more thing" before hitting the sack.

So here you are with *Inside Macintosh*, a tureen of Dark French Roast, and about 15 Tech Notes open, trying to figure out how to get some gadget to work. The feeling of success has been replaced by annoyance. It took 20 minutes of index scouring to find these Tech Notes, and you're still not sure you've got the whole picture.

Writing great software is challenging enough without fighting with documentation. If you take a look at the latest *Developer CD Series* disc, you'll see that Tech Notes and Sample Code have been reorganized; we believe this new system will be far easier for you to use.

The first thing you'll notice is that Tech Notes and Sample Code have new categories; these are based on the ones used in *New Inside Macintosh*, which will be released over the next nine months or so. We want you to be able to quickly collect all the information on a given problem. Take a look at the Categories document to familiarize yourself with the new categories.

Tech Notes have also lost their numbers; the actual document simply has a name. "Numbers have worked fine for me for all these years; why change now?" you ask. Now that the Notes are organized by category, using numbers makes less sense. And using names instead of numbers solves the problem of gaps in the numbering scheme, which happens when Tech Notes are removed. We expect even more of them to be removed than before, because they'll be incorporated into *New Inside Macintosh* periodically as it's revised.

In the Indexes folder, you'll see "alias indexes" that will let you get to the information in several different ways, including by the old Tech Note number. We're trying to cover all the bases.

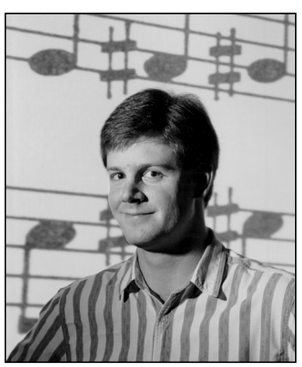
We hope you'll check out the Tech Notes and let us know what you think of the new scheme!

A note for international folks: Tech Notes now print on A4 paper as well as US Letter, so you don't need to spend hours reformatting them by hand! •

Send your feedback on Tech Notes or Sample Code to Neil at AppleLink NMDAY or on the Internet at nmday@apple.com. •

THE ASYNCHRONOUS SOUND HELPER

In system software version 6.0.7 and later, the Sound Manager has impressive sound input and output capabilities that are largely untapped by the existing body of application software. This article presents a code module called the Asynchronous Sound Helper that's designed to make asynchronous sound input and output easily accessible to the application programmer, yet provide an interface flexible enough to facilitate extensive application features.



BRYAN K. ("BEAKER") RESSLER

Of all the Managers in *Inside Macintosh*, the Sound Manager may be the winner of the Most Startling Metamorphosis contest. On the earliest Macintosh computers, sound was produced by direct calls to a Sound Driver, as described in *Inside Macintosh* Volume II. Later, with the advent of system software version 4.1 and the more powerful sound-generation hardware of the Macintosh SE and Macintosh II, the Sound Driver was superseded by a fairly buggy initial implementation of the Sound Manager, which was first documented in *Inside Macintosh* Volume V. The new Sound Manager presented a problem for developers, because there was a large installed base of Macintosh 128K, 512K, 512K enhanced, and Plus computers that didn't have the ROMs or system software to support the Sound Manager. At this point, all but the heartiest developers decided the tradeoffs for including sound in a non–sound-related application were too severe.

By the time version 6.0.7 rolled around, many of the details of the Sound Manager had changed, and sound input support was added. In fact, the Sound Manager in 6.0.7 and System 7 is relatively stable. So if you've been waiting for the right moment to add sound support to your application, the moment has arrived.

The Macintosh Sound Manager acts as a buffer between your application and the complexities of the sound hardware (see Figure 1). Sounds are produced by sending *sound commands* to a *sound channel*. The sound channel sends the commands through a *synthesizer* that knows how to control the audio hardware. Among the Sound Manager's current 38 commands are operations such as playing simple frequencies,

BRYAN K. ("BEAKER") RESSLER (AppleLink: ADOBE.BEAKER) Looking back, it seems clear that Bryan sacrificed quality time with his wife during the writing of this article. So, in the spirit of fairness, *develop* asked Bryan's wife, Nicole, to contribute the bio for her husband. Here it is: "I owe it all to my wife, without whom I wouldn't be the man I am today. The End."•

playing complex recorded sounds, and changing sound volume. The Sound Manager also allows you to record new sounds if the appropriate hardware is available. Recording is performed through a *sound input driver*.

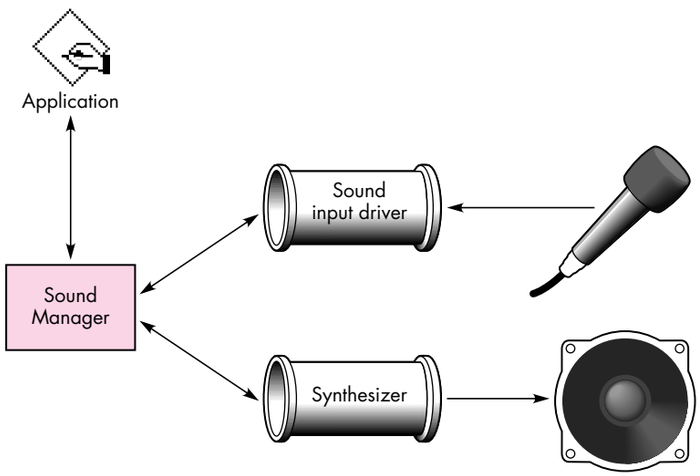


Figure 1
The Sound Manager

Sound playback and recording through the Sound Manager can be performed *synchronously* or *asynchronously*. When you make a synchronous call to the Sound Manager, the function doesn't return control to your application until the entire operation (sound playback, for instance) is complete. In general, it's easy to use the Sound Manager to play or record sound synchronously. Asynchronous calls return control immediately to your application and perform their operations in the background, which makes asynchronous operations somewhat trickier. Many developers feel that there are too many details to make asynchronous sound worthwhile in an application not specifically oriented toward sound. However, with sound input devices becoming more common, the market impetus to add sound is growing.

This article presents the Asynchronous Sound Helper, a code module designed to take much of the heartburn out of asynchronous sound input and output. The goals of Helper, as we'll be calling it from now on, are threefold:

- Provide a straightforward and uncomplicated interface for asynchronous sound I/O specifically tailored toward common application requirements.
- Encourage developers to include support for sound as a standard type of data, just like text or graphics.

- Function as a tutorial on how to perform asynchronous sound input and output using the Sound Manager.

Helper provides two-tiered support—"easy" calls for basic operations and "advanced" calls for more complex operations. You choose which calls to use depending on your application's specific needs and user interface. For simple asynchronous recording and playback, only a few routines are required. Or go all out and use Helper routines to easily provide a "sound palette" with tape-deck–like controls for your application.

To top it off, the overhead for Helper is fairly small. The code compiles to about 4K, and it adds 86 bytes of global data to your application. At run time, it uses around 4K in your application's heap. Helper uses clean Sound Manager techniques—nothing skanky that might cause compatibility problems in the future.

HOOKING UP WITH HELPER

First let's take a quick look at how Helper works and how your application uses it. We'll leave the details for later.

Before you can use Helper you need to add a global Boolean flag to your application —the attention flag. At initialization time, your application calls Helper's initialization routine and provides the address of the attention flag. In its main event loop, your application checks the value of the attention flag and, if true, calls Helper's idle routine.

Because Helper's main function is to spawn asynchronous sound tasks, communication between your application and Helper is carried out on an as-needed basis. Here are the basic phases of communication for a typical sound playback sequence (the numbers correspond to Figure 2).

1. Your application tells Helper to play some sound.
2. Helper uses the Sound Manager to allocate a sound channel and begins asynchronous playback of your sound.
3. The application goes on its merry way, with the sound playing asynchronously in the background.
4. The sound completes playback. Helper has set up a sound command that causes Helper to be informed immediately upon completion of playback (this occurs at interrupt time). At that time, Helper sets the application's global attention flag.
5. The next time through your application's event loop, the application notices that the attention flag is set and calls SHIdle to free up the sound channel.

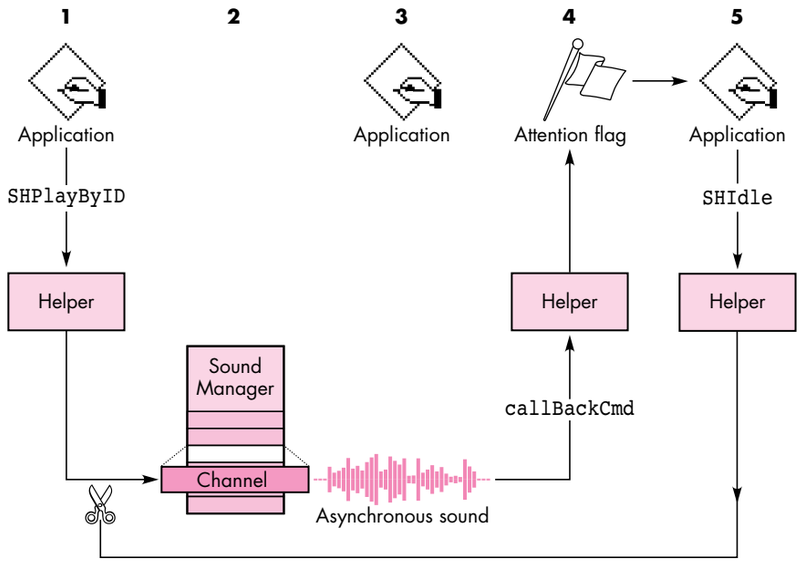


Figure 2
Application-Sound Manager Interface

When your application terminates, it calls Helper's kill routine. Helper's method of communication with the application minimizes processing overhead. By using the attention flag scheme, your application calls Helper's idle routine only when it's really necessary. This could be important in game and multimedia applications where CPU bandwidth is pushed to the limit.

HELPER'S INTERFACE

Now let's take a look at the interfaces and the basic uses of the routines provided by Helper. Later we'll go into more detail about how the routines work and how to use them.

INITIALIZATION, IDLE, AND TERMINATION

```
pascal OSErr SHInitSoundHelper(Boolean *attnFlag, short numChannels);
pascal void SHIdle(void);
pascal void SHKillSoundHelper(void);
```

SHInitSoundHelper initializes Helper. It allocates memory, so you should call it near the beginning of your application. The application passes to SHInitSoundHelper the address of the Boolean attention flag that Helper uses to inform the application when it needs attention.

Helper is written in C, but all public routines are declared as Pascal so that they can be called from other languages.•

SHIdle performs various cleanup tasks. Call SHIdle when the attention flag goes true.

At application termination, call SHKillSoundHelper. It stops current recording and playback and deallocates Helper's memory.

EASY SOUND OUTPUT

```
pascal OSErr SHPlayByID(short resID, long *refNum);
pascal OSErr SHPlayByHandle(Handle sound, long *refNum);
pascal OSErr SHPlayStop(long refNum);
pascal OSErr SHPlayStopAll(void);
```

SHPlayByID and SHPlayByHandle provide an easy way to begin asynchronous sound playback. These routines return a reference number via the refNum parameter. This reference number can be used to stop playback and can be used with the advanced routines described later. If you intend to simply trigger a sound that you want to run to completion (like a gunshot sound in a game), you can pass nil for the refNum parameter, thereby ignoring the reference number.

To stop a given sound or stop all playback, use SHPlayStop or SHPlayStopAll.

ADVANCED SOUND OUTPUT

```
pascal OSErr SHPlayPause(long refNum);
pascal OSErr SHPlayContinue(long refNum);
pascal SHPlayStat SHPlayStatus(long refNum);
pascal OSErr SHGetChannel(long refNum, SndChannelPtr *channel);
```

If you want more control over the playback process, these routines will be of interest. SHPlayPause pauses the playback of a sound, like the pause button on a tape deck. SHPlayContinue continues playback of a sound that was previously paused. Use SHPlayStatus to find out the status of a sound—finished, playing, or paused. If you want to send commands directly to a sound channel that was allocated by Helper, use SHGetChannel. You might want to send sound commands in your application, for example, to play continuous looped background music.

EASY SOUND INPUT

```
pascal OSErr SHRecordStart(short maxK, OSType quality, Boolean *doneFlag);
pascal OSErr SHGetRecordedSound(Handle *theSound);
pascal OSErr SHRecordStop(void);
```

These are the three basic routines for recording sound through a sound input device. To begin asynchronous sound recording, use SHRecordStart. The application passes the address of a Boolean—a recording-completed flag—that tells the application

when the recording is complete. Once complete, the application calls SHGetRecordedSound to retrieve a sound handle. The handle is suitable for playback with SHPlayByHandle or to be written out as a 'snd ' resource. To stop recording immediately (as with the stop button on a tape recorder), use SHRecordStop.

ADVANCED SOUND INPUT

```
pascal OSErr SHRecordPause(void);
pascal OSErr SHRecordContinue(void);
pascal OSErr SHRecordStatus(SHRecordStatusRec *recordStatus);
```

To pause recording, use SHRecordPause. To continue previously paused recording, use SHRecordContinue. Use SHRecordStatus to get information about the status of recording. This status information includes the current input level (which could be used to draw a tape-deck–like level meter), the amount of sound that's been recorded (with respect to the maximum), and whether the recording is finished, recording, or paused.

HELPER'S DATA STRUCTURES

Helper uses three internal data structures to keep track of recording and playback.

```
typedef struct {
   short       numOutRecs;    // The number of output records in outArray.
   SHOutRec    *outArray;     // Our preallocated output records.
   long        nextRef;       // Next available output reference number.
} SHOutputVars;
```

The SHOutputVars record contains an array of SHOutRec records. The numOutRecs field tells how many are in the array. These records, one for each allocated channel, hold information about currently playing sounds. They're reused when sounds have completed. The SHOutputVars record also keeps track of the next available output reference number, in the field nextRef. The reference numbers are unique (modulo 2,147,483,647).

```
typedef struct {
   SndChannel  channel;       // Our sound channel.
   long        refNum;        // Our Helper reference number.
   Handle      sound;         // The sound we're playing.
   Fixed       rate;          // Sampled sound playback rate.
   char        handleState;   // The handle state for this handle.
   Boolean     inUse;         // Tells whether this SHOutRec is in use.
   Boolean     paused;        // Tells whether this sound is paused.
} SHOutRec, *SHOutPtr;
```

An SHOutRec record's first field, channel, contains the actual Sound Manager SndChannel used to play the sound. The sound reference number associated with this sound (the one passed back to the application) is stored in the refNum field. A handle to the sound we're playing is stored in the sound field. The rate field holds the sample playback rate of sampled sounds, which is used when pausing sampled sounds. The handleState field contains the original handle state (derived via a call to SHGetState), so Helper can reset the handle's state after playback is complete. The inUse field tells whether a given SHOutRec is in use by a playing sound (as opposed to available for reuse). Finally, the paused flag lets Helper remember when a sound has been paused.

```
typedef struct {
   long        inRefNum;         // Sound Manager's input device refNum.
   SPB         inPB;             // The input parameter block.
   Handle      inHandle;         // The handle we're recording into.
   short       headerLength;     // The length of the sound's header.
   Boolean     recording;        // Tells that we're actually recording.
   Boolean     recordComplete;   // Tells that recording is complete.
   OSErr       recordErr;        // Error, if error terminated recording.
   short       numChannels;      // Number of channels for recording.
   short       sampleSize;       // Sample size for recording.
   Fixed       sampleRate;       // Sample rate for recording.
   OSType      compType;         // Compression type for recording.
   Boolean     *appComplete;     // Tells caller when recording is done.
   Boolean     paused;           // Tells that recording has been paused.
} SHInputVars;
```

The SHInputVars record contains information pertaining to a sound being recorded. When the sound input device is opened, its reference number is stored in inRefNum. The sound input parameter block, inPB, is part of SHInputVars. The sound being recorded is stored in inHandle until complete. The recording flag tells whether we're actually in the act of asynchronous recording, and the recordComplete flag (set by the record completion routine, described later) tells us when recording has completed. If an error occurs during recording, it's saved in recordErr so that it can be returned to the application later, when it calls SHGetRecordedSound. The next four fields—numChannels, sampleSize, sampleRate, and compType—hold information that's used to construct the sound's header. The appComplete field points to a Boolean that the application may optionally use to be informed of recording termination (the application may use repeated calls to the SHRecordStatus routine instead). The paused flag lets Helper keep track of when recording has been paused.

Helper declares its global storage as shown on the following page. As we go on, you'll see the use of these globals in context, which will clarify their function.

```
static Boolean       gsSHInited = false;  // Has Helper been initialized?
static Boolean       *gsSHNeedsTime;      // Pointer to application's
                                          // attention flag.
static SHOutputVars  gsSHOutVars;         // Sound output variables.
static SHInputVars   gsSHInVars;          // Sound input variables.
```

HELPER'S INTERNAL UTILITY ROUTINES

Helper uses twelve static utility routines to help it carry out its job. Many of these routines are trivial, but let's go over a few of the more important ones in detail—SHPlayCompletion, SHRecordCompletion, and SHOutRecFromRefNum.

When Helper performs asynchronous sound playback, it depends on a callback routine that signals to the application that playback has completed. Here's the playback callback routine, SHPlayCompletion:

```
pascal void SHPlayCompletion(SndChannelPtr channel, SndCommand *command)
{
   long  otherA5;

   // Look for our "callback signature" in the sound command.
   if (command->param1 == kSHCompleteSig) {
      otherA5 = SetA5(command->param2);   // Set up our A5.

      channel->userInfo = kSHComplete;
      *gsSHNeedsTime = true;              // Tell application to give us
                                          // an SHIdle call.

      SetA5(otherA5);                     // Restore old A5.
   }
}
```

When Helper begins the sound playback, it queues up a sound command—a callBackCmd—in the sound channel. The callBackCmd tells the Sound Manager to call the callback routine, SHPlayCompletion. We place a verifiable "signature" in the sound command record so that the application can verify that the call occurred as a result of a specific callBackCmd, and not as a result of some spurious one. When such a blessed callback occurs, Helper uses another handy value stuffed into the sound command—a pointer to the A5 global world—to set up access to the globals. Helper then sets the channel's userInfo field to a value that flags the sound as complete. Helper also sets the application's attention flag so that later, in the main event loop, the application sees that the attention flag is set and calls SHIdle. SHIdle then skips through the SHOutRec array looking for sound channels that are in use and have kSHComplete in their userInfo fields, and disposes of their sound channels. This is how Helper cleans up after sound playback has completed.

Asynchronous sound recording also relies on a callback routine that signals when recording has completed. Here's the callback routine, SHRecordCompletion:

```
pascal void SHRecordCompletion(SPBPtr inParams)
{
   long  otherA5;

   otherA5 = SetA5(inParams->userLong);   // Set up our A5.

   *gsSHNeedsTime = true;                 // Tell application to give us
                                          // an SHIdle call.
   gsSHInVars.recordComplete = true;      // Make a note to ourselves.

   SetA5(otherA5);                        // Restore old A5.
}
```

When recording has completed (for any reason—we filled the entire buffer, an error occurred, or the user manually stopped recording), the Sound Manager calls the record callback routine. Like the playback callback routine, it first sets up the A5 world. Then it sets the application's attention flag and the recordComplete flag inside the global SHInputVars structure. Later, the application will notice its attention flag is set and call SHIdle. SHIdle checks the recordComplete flag and notices that recording is complete, closes the sound input device, and prepares for the application to call SHGetRecordedSound to retrieve the recorded sound. This is how Helper cleans up after asynchronous sound recording.

Another heavily used static utility routine is SHOutRecFromRefNum. It maps a sound reference number into a pointer to the appropriate SHOutRec.

```
SHOutPtr SHOutRecFromRefNum(long refNum)
{
   short i;

   // Search for the specified refNum.
   for (i = 0; i < gsSHOutVars.numOutRecs; i++)
      if (gsSHOutVars.outArray[i].inUse &&
            gsSHOutVars.outArray[i].refNum == refNum)
         break;

   // If we found it, return a pointer to that record, otherwise nil.
   if (i == gsSHOutVars.numOutRecs)
      return(nil);
   else return(&gsSHOutVars.outArray[i]);
}
```

SHOutRecFromRefNum simply does a linear search through the output records, looking for a record that is in use and has a matching reference number. If none is found, nil is returned.

We'll investigate a few more utility routines as we delve into the details of the public routines in the sections that follow.

HELPER'S INITIALIZATION, TERMINATION, AND IDLE ROUTINES

Let's take a closer look at the SHInitSoundHelper, SHKillSoundHelper, and SHIdle routines.

SHINITSOUNDHELPER

```
pascal OSErr SHInitSoundHelper(Boolean *attnFlag, short numChannels)
{
   OSErr err;

   // Use default number of channels if zero was specified.
   if (numChannels == 0)
      numChannels = kSHDefChannels;

   // Remember the address of the application's attention flag.
   gsSHNeedsTime = attnFlag;

   // Allocate the channels.
   gsSHOutVars.numOutRecs = numChannels;
   gsSHOutVars.outArray = (SHOutPtr)NewPtrClear(numChannels *
      sizeof(SHOutRec));

   // If successful, flag that we're initialized and exit.
   if (gsSHOutVars.outArray != nil) {
      gsSHInited = true;
      return(noErr);
   } else {
      // Return some kind of error (MemError if there is one, otherwise
      // make one up).
      err = MemError();
      if (err == noErr)
         err = memFullErr;
      return(err);
   }
}
```


A note to MacApp users: Set PermAllocation to true before calling SHInitSoundHelper; otherwise the outArray pointer may be allocated from temporary storage.•

SHInitSoundHelper is fairly uncomplicated. The attnFlag parameter points to the application's attention flag, which is used to tell the application that a call to SHIdle is needed. The numChannels parameter tells Helper how many channels to allocate. The number of simultaneous sounds that can be played back by Helper is limited by the number of channels allocated (via numChannels) and the number of simultaneous sound channels the Sound Manager allows. So use a numChannels that's appropriate to your needs. If you specify zero, a reasonable default (four) is used. SHInitSoundHelper allocates the output records and stores a pointer to the array in gsSHOutVars. If the memory allocation is successful, gsSHInited is set to true.

SHKILLSOUNDHELPER

```
pascal void SHKillSoundHelper(void)
{
   short    i;
   long     timeout;
   Boolean  outputClean, inputClean;

   if (!gsSHInited)
      return;

   SHPlayStopAll();     // Kill all playback.
   SHRecordStop();      // Kill recording.

   // Now sync-wait for everything to clean itself up.
   timeout = TickCount() + kSHSyncWaitTimeout;
   do {
      if (*gsSHNeedsTime)
         SHIdle();      // Clean up when required.

      // Check whether all our output channels are cleaned up.
      outputClean = true;
      for (i = 0; i < gsSHOutVars.numOutRecs && outputClean; i++)
         if (gsSHOutVars.outArray[i].inUse)
            outputClean = false;

      // Check whether our recording is cleaned up.
      inputClean = !gsSHInVars.recording;

      if (inputClean && outputClean)
         break;
   } while (TickCount() < timeout);

   // Lose our preallocated sound channels.
   DisposePtr((Ptr)gsSHOutVars.outArray);
}
```

SHKillSoundHelper first stops any asynchronous sound input or output in progress. It waits for all the output channels to be free and for recording to stop before continuing. Finally, it disposes of the output record array.

SHIDLE

```
pascal void SHIdle(void)
{
   short i;
   OSErr err;
   long  realSize;

   // Immediately turn off the application's attention flag.
   *gsSHNeedsTime = false;

   // Do playback cleanup.
   for (i = 0; i < gsSHOutVars.numOutRecs; i++)
      if (gsSHOutVars.outArray[i].inUse &&
            gsSHOutVars.outArray[i].channel.userInfo == kSHComplete)
         // We've found a channel that needs closing.
         SHReleaseOutRec(&gsSHOutVars.outArray[i]);

   // Do recording cleanup.
   if (gsSHInVars.recording && gsSHInVars.recordComplete) {
      HUnlock(gsSHInVars.inHandle);
      if (gsSHInVars.inPB.error && gsSHInVars.inPB.error != abortErr) {
         // An error (other than a manual stop) occurred during
         // recording. Kill the handle and save the error code.
         gsSHInVars.recordErr = gsSHInVars.inPB.error;
         DisposeHandle(gsSHInVars.inHandle);
         gsSHInVars.inHandle = nil;
      } else {
         // Recording completed normally (which includes abortErr, the
         // "error" that occurs when recording is stopped manually).
         gsSHInVars.recordErr = noErr;
         realSize = gsSHInVars.inPB.count + gsSHInVars.headerLength;
         err = SetupSndHeader(gsSHInVars.inHandle, gsSHInVars.numChannels,
            gsSHInVars.sampleRate, gsSHInVars.sampleSize,
            gsSHInVars.compType, kSHBaseNote, realSize,
            &gsSHInVars.headerLength);
         SetHandleSize(gsSHInVars.inHandle, realSize);
      }

      // Error or not, close the recording device and tell the application
      // that recording is complete through the recording-completed
      // flag that the caller originally passed into SHRecordStart.
```

```
      SPBCloseDevice(gsSHInVars.inRefNum);
      gsSHInVars.recording = false;
      gsSHInVars.inRefNum = 0;
      if (gsSHInVars.appComplete != nil)
         *gsSHInVars.appComplete = true;
   }
}
```

SHIdle is one of the most important routines in Helper. It performs cleanup of completed sound playback and recording. First SHIdle clears the application's attention flag. For playback cleanup, it iterates through the output records looking for records that have their inUse flag set and have kSHComplete in their sound channel's userInfo field. These sounds have been flagged as completed by the callback routine. When such an output record is found, its channel is closed with a call to SHReleaseOutRec.

```
void SHReleaseOutRec(SHOutPtr outRec)
{
   short    i;
   Boolean  found = false;

   // An SHOutRec's inUse flag gets set only if SndNewChannel has been
   // called on the record's sound channel. So if it's in use, we call
   // SndDisposeChannel and ignore the error. (What else can we do?)
   if (outRec->inUse)
      SndDisposeChannel(&outRec->channel, kSHQuietNow);

   // If this sound handle isn't being used by some other output record,
   // kindly restore the original handle state.
   if (outRec->sound != nil) {
      for (i = 0; i < gsSHOutVars.numOutRecs && !found; i++)
         if (&gsSHOutVars.outArray[i] != outRec &&
               gsSHOutVars.outArray[i].inUse &&
               gsSHOutVars.outArray[i].sound == outRec->sound)
            found = true;

      if (!found)
         HSetState(outRec->sound, outRec->handleState);
   }

   outRec->inUse = false;
}
```

The SHReleaseOutRec routine has two important functions. First, it calls SndDisposeChannel to free up the sound channel. Second, it restores the handle state

of the sound that was playing if that same sound isn't currently playing on some other channel.

Recording cleanup is also performed back in SHIdle. If the recording flag *and* the recordComplete flag are set, the record callback has informed Helper that recording is complete. Right away, Helper unlocks the sound handle. Next Helper checks for errors. If the application called SHRecordStop to manually stop recording before the buffer was full, the error abortErr is generated. We don't really consider this an error, so we expressly allow abortErr. If an error did occur, Helper saves the error code. This way, later, when the application calls SHGetRecordedSound, Helper can return an appropriate OSErr. If no error occurred, Helper calculates the actual size of the sampled sound and builds an appropriate sound header, including the correct length.

After checking for errors, Helper resizes the handle to exactly the size it should be. Then it calls SPBCloseDevice to close the sound input device, clears the recording flag, and sets the application's recording-completed flag, if one was provided.

As you can see, it's important to call SHIdle when the attention flag goes true; otherwise subsequent requests for playback or recording may fail.

EASY PLAYBACK ROUTINES

Now we'll look more closely at Helper's easy playback routines, SHPlayByID, SHPlayByHandle, SHPlayStop, and SHPlayStopAll.

SHPLAYBYID AND SHPLAYBYHANDLE

```
pascal OSErr SHPlayByID(short resID, long *refNum)
{
   Handle   sound;
   char     oldHandleState;
   short    ref;
   OSErr    err;
   SHOutPtr outRec;

   // First, try to get the caller's 'snd ' resource.
   sound = GetResource(soundListRsrc, resID);
   if (sound == nil) {
      err = ResError();
      if (err == noErr)
         err = resNotFound;
      return(err);
   }
   oldHandleState = SHGetState(sound);
   HNoPurge(sound);
```

```
   // Now let's get a reference number and an output record.
   ref = SHNewRefNum();
   err = SHNewOutRec(&outRec);
   if (err != noErr) {
      HSetState(sound, oldHandleState);
      return(err);
   }

   // Now let's fill in the output record. This routine also initializes
   // the sound channel and flags outRec as "in use."
   err = SHInitOutRec(outRec, ref, sound, oldHandleState);
   if (err != noErr) {
      HSetState(sound, oldHandleState);
      SHReleaseOutRec(outRec);
      return(err);
   }

   // We're in pretty good shape. We've got a reference number, an
   // initialized output record, and the sound handle. Let's party.
   MoveHHi(sound);
   HLock(sound);
   err = SHBeginPlayback(outRec);
   if (err != noErr) {
      HSetState(sound, oldHandleState);
      SHReleaseOutRec(outRec);
      return(err);
   } else {
      if (refNum != nil)   // refNum is optional--the caller may not
         *refNum = ref;    // want it.
      return(noErr);
   }
}
```

SHPlayByID starts asynchronous playback of the 'snd ' resource with ID resID. First the resource is loaded and set to be nonpurgeable. Notice the call to SHGetState. This utility routine searches the output record array looking for the given sound handle in some output record that's flagged as inUse. If the handle is found, SHGetState returns the handle state that's stored in the output record. If the sound handle isn't found, the function returns HGetState(sound). See "Why SHGetState?" for details on why this is necessary.

Then SHPlayByID calls SHNewRefNum to get the next consecutive sound reference number, and SHNewOutRec to find the first available output record in the list. Next, SHPlayByID calls SHInitOutRec to fill out the output record.

WHY SHGETSTATE?

SHGetState is necessary because your application might trigger the same sound handle twice, the second time while the first is still playing. If SHPlayByID used only HGetState, here's what would happen:

1. At time t_0 your application calls SHPlayByID. The handle's state is retrieved—unlocked and purgeable—and stored in output record 0. So far, all is well, and the sound begins playing.
2. Later, at time t_1, your application makes a new call to SHPlayByID to trigger the same sound again while the first call is still playing. SHPlayByID calls HGetState to get the handle's state—locked, nonpurgeable—and stores it in output record 1 (perhaps you see the problem already). The sound begins playing a second time, over the one that's already playing.
3. At time t_2, the first sound completes. Your application's attention flag gets set, and you dutifully call SHIdle. SHIdle retrieves the sound's original state—unlocked and purgeable—from output record 0 and sets the sound handle to that state.
4. At time t_3, the second sound completes. Again, SHIdle sets the sound handle's state according to what's stored in the output record—locked and nonpurgeable.

We're left with the sound handle in the wrong state. So instead of HGetState, SHPlayByID uses SHGetState. SHGetState looks to see if the sound has already been triggered, and if so, returns the state stored in the previous trigger's output record. Also, SHReleaseOutRec doesn't reset the handle's state if the sound handle is found to be currently playing on some other channel.

```
OSErr SHInitOutRec(SHOutPtr outRec, long refNum, Handle sound,
   char handleState)
{
   short          i;
   OSErr          err;
   SndChannelPtr  channel;

   // Initialize the sound channel inside outRec. Clear the bytes to
   // zero, install the proper queue size, and then call SndNewChannel.
   for (i = 0; i < sizeof(SndChannel); i++)
      ((char *)&outRec->channel)[i] = 0;
   outRec->channel.qLength = stdQLength;
   channel = &outRec->channel;
   err = SndNewChannel(&channel, kSHNoSynth, kSHNoInit,
      (SndCallBackProcPtr)SHPlayCompletion);
   if (err != noErr)
      return(err);

   // Initialize the rest of the record and return noErr. Note that we
   // set the record's inUse flag only if the SndNewChannel call was
   // successful.
   outRec->refNum = refNum;
   outRec->sound = sound;
```

```
   outRec->rate = 0;
   outRec->handleState = handleState;
   outRec->inUse = true;
   outRec->paused = false;
   return(noErr);
}
```

The SHInitOutRec routine calls SndNewChannel to open the sound channel that's associated with this output record. The constant kSHNoSynth is passed as the synthesizer and kSHNoInit is passed as the synthesizer initializer value. These values are passed because Helper doesn't have any idea what kind of sound will be played on this channel, so it must assume nothing. (See "Types of Sound" for an overview of the different synthesizers.) SHInitOutRec also passes the address of the playback completion routine, SHPlayCompletion, to SndNewChannel. If successful, the rest of the output record is filled out and the output record's inUse flag is set.

If the SHInitOutRec call is successful, SHPlayByID moves the handle high in the heap, locks it, and begins playback with a call to SHBeginPlayback.

```
OSErr SHBeginPlayback(SHOutPtr outRec)
{
   OSErr err;

   // First, initiate playback. If an error occurs, return it
   // immediately.
   err = SndPlay(&outRec->channel, outRec->sound, kSHAsync);
   if (err != noErr)
      return(err);

   // Playback started OK. Let's queue up a callback command so that
   // we'll know when the sound is finished.
   SHQueueCallback(&outRec->channel);  // Ignore error. (What can we do?)
   return(noErr);
}
```

The SHBeginPlayback routine calls SndPlay to start the sound playing asynchronously, passing as parameters the sound handle and the flag kSHAsync. Since the only way to tell that an asynchronous sound has completed is via a callback, Helper must queue up a callBackCmd after beginning playback. This is done with a call to SHQueueCallback.

Finally, SHPlayByID returns the sound reference number, if the application wants it. (You can pass nil if you don't care about the reference number.)

TYPES OF SOUND

The Sound Manager supports three basic types of sound. First is simple *square-wave synthesis*. You can specify the amplitude (volume), frequency (pitch), approximate timbre, and duration of sounds for a square-wave synthesizer with the Sound Manager commands ampCmd, timbreCmd, and freqDurationCmd.

The second type of sound is *wave-table synthesis*, which allows you to specify a waveform as 512 *samples*. These samples specify the relative output voltage over time for one period of the waveform. Sounds with more complex timbre can be created using a wave-table synthesizer. You control the frequency and amplitude of wave-table sound in the same way as square-wave sound.

The most interesting sounds can be produced via the third type—*sampled synthesis*. Sampled sounds are a continuous list of relative voltages over time that allow the Sound Manager to reconstruct an arbitrary analog waveform. This could be a recording of music, your voice —anything.

Helper allows you to easily play any of these types of sound asynchronously.

The SHPlayByHandle routine is similar to SHPlayByID, except that it supports a special case: you can pass SHPlayByHandle a nil handle. This means "go ahead and open a sound channel, but don't call SndPlay." Normally an application that does this subsequently calls SHGetChannel to retrieve the sound channel pointer and sends sound commands directly to the channel itself. This is covered in more detail later in the section "Advanced Playback Routines."

SHPLAYSTOP AND SHPLAYSTOPALL

```
pascal OSErr SHPlayStop(long refNum)
{
   SHOutPtr outRec;

   // Look for the associated output record.
   outRec = SHOutRecFromRefNum(refNum);

   // If we found it, call SHPlayStopByRec to stop playback.
   if (outRec != nil) {
      SHPlayStopByRec(outRec);
      return(noErr);
   } else return(kSHErrBadRefNum);
}
```

SHPlayStop stops playback of a given sound by looking up the reference number. The routine tries to find the output record associated with refNum by a call to SHOutRecFromRefNum. If one is found, SHPlayStop calls SHPlayStopByRec to do the actual work.

```
pascal OSErr SHPlayStopAll(void)
{
   short i;

   // Look for output records that are in use and stop their playback
   // with SHPlayStopByRec.
   for (i = 0; i < gsSHOutVars.numOutRecs; i++)
      if (gsSHOutVars.outArray[i].inUse)
         SHPlayStopByRec(&gsSHOutVars.outArray[i]);

   return(noErr);
}
```

SHPlayStopAll is not much different, but instead of looking up a reference number, it calls SHPlayStopByRec on all output records that have their inUse flag set. Let's take a look at SHPlayStopByRec.

```
void SHPlayStopByRec(SHOutPtr outRec)
{
   SndCommand  cmd;

   // Dump the rest of the commands in the queue (including our
   // callBackCmd).
   cmd.cmd = flushCmd;
   cmd.param1 = 0;
   cmd.param2 = 0;
   SndDoImmediate(&outRec->channel, &cmd);

   // Shut up and go to your room! No dessert for you, little boy.
   cmd.cmd = quietCmd;
   cmd.param1 = 0;
   cmd.param2 = 0;
   SndDoImmediate(&outRec->channel, &cmd);

   // It's now safe to manually dump our channel (we'll just skip the
   // whole callback thing in this case).
   SHReleaseOutRec(outRec);
}
```

To stop a playing sound, Helper sends a flushCmd, which flushes all subsequent (currently unprocessed) sound commands from a channel's queue, and a quietCmd, which tells the channel to stop making sound. The flushCmd also flushes the callBackCmd we previously queued up. After these two commands, we can safely call SHReleaseOutRec to dispose of the sound channel for the sound.

Now that we've seen the basic stuff, on to the advanced sound output routines.

ADVANCED PLAYBACK ROUTINES

Helper's easy calls are enough to satisfy the demands of many applications. If finer control is desired, a few other playback routines can be used. Let's take a closer look at the advanced playback routines, SHPlayPause, SHPlayContinue, SHPlayStatus, and SHGetChannel.

SHPLAYPAUSE

```
pascal OSErr SHPlayPause(long refNum)
{
   SHOutPtr    outRec;
   SndCommand  cmd;
   OSErr       err;

   outRec = SHOutRecFromRefNum(refNum);
   if (outRec != nil) {
      // Don't bother with this if we're already paused.
      if (outRec->paused)
         return(kSHErrAlreadyPaused);

      // Get the current playback rate for this sound.
      cmd.cmd = getRateCmd;
      cmd.param1 = 0;
      cmd.param2 = &outRec->rate;
      err = SndDoImmediate(&outRec->channel, &cmd);
      if (err != noErr)
         return(err);

      // Now pause with either a rateCmd or a pauseCmd, as appropriate.
      cmd.param1 = 0;
      cmd.param2 = 0;
      if (outRec->rate != 0) {
         // If we get something nonzero, it's safe to assume that
         // whatever synthesizer we're talking to will be able to
         // understand a rateCmd to restore the rate (probably the
         // sampled synthesizer). To pause the sound, we'll set the
         // rate to zero.
         cmd.cmd = rateCmd;
         err = SndDoImmediate(&outRec->channel, &cmd);
         if (err != noErr)
            return(err);
```

```
      } else {
         // This synthesizer doesn't understand rateCmds. So instead
         // we'll just pause command queue processing with a pauseCmd.
         // This is how we pause command-type sounds (e.g., Simple Beep).
         cmd.cmd = pauseCmd;
         err = SndDoImmediate(&outRec->channel, &cmd);
         if (err != noErr)
            return(err);
      }

      outRec->paused = true;
      return(noErr);
   } else return(kSHErrBadRefNum);
}
```

There are two basic methods of pausing a sound: one uses a pauseCmd, the other uses a rateCmd. Sounds that are composed of a lot of little sound commands (like Simple Beep) are paused by pausing command-queue processing with a pauseCmd. Most sampled sounds, however, have only one command, a bufferCmd, which plays the sampled sound. A pauseCmd is ineffective for this type of sound because it pauses command-queue processing after the completion of the bufferCmd; in essence, the sound plays to completion before pausing. Therefore, a different approach is taken with sampled sounds: a rateCmd is used to set the sample playback rate to 0.0, effectively stopping the bufferCmd in its tracks.

SHPlayPause first retrieves the output record associated with the given refNum, and then checks that the sound is not already paused. SHPlayPause then sends a getRateCmd to establish the current playback rate of the sound. If getRateCmd returns a nonzero rate, SHPlayPause knows that a rateCmd can be used to pause the sound; otherwise a pauseCmd is used. Either way, SHPlayPause sets the paused flag in the output record.

SHPLAYCONTINUE

```
pascal OSErr SHPlayContinue(long refNum)
{
   SHOutPtr    outRec;
   SndCommand  cmd;
   OSErr       err;

   outRec = SHOutRecFromRefNum(refNum);
   if (outRec != nil) {
      // Don't even bother with this stuff if the channel isn't paused.
      if (!outRec->paused)
         return(kSHErrAlreadyContinued);
```

```
      // Now continue playback with a rateCmd or a resumeCmd, as
      // appropriate.
      cmd.param1 = 0;
      if (outRec->rate != 0) {
         // Resume sampled sound playback by restoring the synthesizer's
         // playback rate with a rateCmd.
         cmd.cmd = rateCmd;
         cmd.param2 = outRec->rate;
         err = SndDoImmediate(&outRec->channel, &cmd);
         if (err != noErr)
            return(err);
      } else {
         // Resume sound queue processing with a resumeCmd.
         cmd.cmd = resumeCmd;
         cmd.param2 = 0;
         err = SndDoImmediate(&outRec->channel, &cmd);
         if (err != noErr)
            return(err);
      }

      outRec->paused = false;
      return(noErr);
   } else return(kSHErrBadRefNum);
}
```

SHPlayContinue continues the playback of a previously paused sound, checking whether there's a nonzero rate in the output record. This is the indicator of whether to send a resumeCmd or rateCmd. If the rate is zero, SHPlayContinue sends a resumeCmd to resume the sound. If the rate is nonzero, SHPlayContinue sends a rateCmd to restore the sample playback rate for the sound.

SHPLAYSTATUS

```
pascal SHPlayStat SHPlayStatus(long refNum)
{
   SHOutPtr outRec;

   if (refNum >= gsSHOutVars.nextRef)
      return(shpError);
   else {
      outRec = SHOutRecFromRefNum(refNum);

      if (outRec != nil) {
         // We found an SHOutRec for the refNum (so it's in use).
         return((outRec->paused) ? shpPaused : shpPlaying);
```

```
      } else {
         // Although we've used the reference number in the past,
         // it's not in use, so we can assume whatever sound it was
         // associated with has stopped. Therefore, we'll return
         // shpFinished.
         return(shpFinished);
      }
   }
}
```

SHPlayStatus returns status information about a given sound, by reference number. The SHPlayStat enum looks like this:

```
typedef enum {
   shpError = -1,
   shpFinished = 0,
   shpPaused = 1,
   shpPlaying = 2
} SHPlayStat;
```

SHPlayStatus uses the fact that sound reference numbers are sequential and unique to infer the status of a sound, even if its record is no longer in the output record array. If refNum is greater than the next available reference number, SHPlayStatus returns shpError, since refNum is invalid. If refNum can be found in the output record list, SHPlayStatus returns shpPlaying or shpPaused, depending on the state of the output record's paused flag. And finally, if refNum is not in use by an existing output record but has been used in the past, it's safe to assume that playback has completed for that reference number, and SHPlayStatus returns shpFinished.

SHGETCHANNEL

Finally, there's SHGetChannel. This routine allows you to use Helper to do sound channel management but retain the ability to send sound commands to the channel yourself. This is most commonly done to play looped continuous music in the background.

To use Helper to play looped continuous music, the application calls the SHPlayByHandle routine with nil as the sound handle. This tells Helper to open the channel without a subsequent call to SndPlay. Then the application calls SHGetChannel to retrieve a pointer to the sound channel that Helper has set up. The application loads a sound resource containing a soundCmd, which installs a sampled sound as a voice. It plays this sound in the channel with a call to SndPlay, then issues a freqCmd to start it playing indefinitely. The demonstration program SHDemo provided on the *Developer CD Series* disc gives a specific example of this technique.

```
pascal OSErr SHGetChannel(long refNum, SndChannelPtr *channel)
{
   SHOutPtr outRec;

   // Look for the output record associated with refNum.
   outRec = SHOutRecFromRefNum(refNum);

   // If we found one, return a pointer to the sound channel.
   if (outRec != nil) {
      *channel = &outRec->channel;
      return(noErr);
   } else return(kSHErrBadRefNum);
}
```

SHGetChannel simply searches for the output record associated with refNum. If one is found, a pointer to the sound channel is returned via the channel parameter.

EASY RECORDING ROUTINES

Helper provides routines to simplify the process of asynchronous sound recording. Most applications' needs will be satisfied by the three easy routines, SHRecordStart, SHGetRecordedSound, and SHRecordStop.

```
pascal OSErr SHRecordStart(short maxK, OSType quality, Boolean *doneFlag)
{
   Boolean  deviceOpened = false;
   Boolean  allocated = false;

   OSErr    err;
   short    canDoAsync;
   short    metering;
   long     allocSize;

   // 1. Try to open the current sound input device.
   err = SPBOpenDevice(nil, siWritePermission, &gsSHInVars.inRefNum);
   if (err == noErr)
      deviceOpened = true;

   // 2. Now let's see if this device can handle asynchronous recording.
   if (err == noErr) {
      err = SPBGetDeviceInfo(gsSHInVars.inRefNum, siAsync,
         (Ptr)&canDoAsync);
      if (err == noErr && !canDoAsync)
         err = kSHErrNonAsyncDevice;
   }
```

```
// 3. Try to allocate memory for the application's sound.
if (err == noErr) {
   allocSize = (maxK * 1024) + kSHHeaderSlop;
   gsSHInVars.inHandle = NewHandle(allocSize);
   if (gsSHInVars.inHandle == nil) {
      err = MemError();
      if (err == noErr)
         err = memFullErr;
   }
   if (err == noErr)
      allocated = true;
}

// 4. Set up various recording parameters (metering and quality).
if (err == noErr) {
   metering = 1;
   SPBSetDeviceInfo(gsSHInVars.inRefNum, siLevelMeterOnOff,
      (Ptr)&metering);
   err = SPBSetDeviceInfo(gsSHInVars.inRefNum, siRecordingQuality,
      (Ptr)&quality);
}

// 5. Call SHGetDeviceSettings to determine a bunch of information
// we'll need to make a header for this sound.
if (err == noErr) {
   err = SHGetDeviceSettings(gsSHInVars.inRefNum,
      &gsSHInVars.numChannels, &gsSHInVars.sampleRate,
      &gsSHInVars.sampleSize, &gsSHInVars.compType);
}

// 6. Create a header for this sound.
if (err == noErr) {
   err = SetupSndHeader(gsSHInVars.inHandle, gsSHInVars.numChannels,
      gsSHInVars.sampleRate, gsSHInVars.sampleSize,
      gsSHInVars.compType, kSHBaseNote, allocSize,
      &gsSHInVars.headerLength);
}

// 7. Lock the input sound handle and set up the input parameter
// block.
if (err == noErr) {
   MoveHHi(gsSHInVars.inHandle);
   HLock(gsSHInVars.inHandle);
   allocSize -= gsSHInVars.headerLength;
   gsSHInVars.inPB.inRefNum = gsSHInVars.inRefNum;
```

```
      gsSHInVars.inPB.count = allocSize;
      gsSHInVars.inPB.milliseconds = 0;
      gsSHInVars.inPB.bufferLength = allocSize;
      gsSHInVars.inPB.bufferPtr = *gsSHInVars.inHandle +
         gsSHInVars.headerLength;
      gsSHInVars.inPB.completionRoutine = (ProcPtr)SHRecordCompletion;
      gsSHInVars.inPB.interruptRoutine = nil;
      gsSHInVars.inPB.userLong = SetCurrentA5();   // For our
                                                   // completion routine.
      gsSHInVars.inPB.error = noErr;
      gsSHInVars.inPB.unused1 = 0;

      err = noErr;
   }

   // 8. Finally, if all went well, set our recording flag, make sure our
   // recording-completed flag is clear, and initiate asynchronous
   // recording.
   if (err == noErr) {
      gsSHInVars.recording = true;
      gsSHInVars.recordComplete = false;
      gsSHInVars.appComplete = doneFlag;
      gsSHInVars.paused = false;
      if (gsSHInVars.appComplete != nil)
         *gsSHInVars.appComplete = false;

      err = SPBRecord(&gsSHInVars.inPB, kSHAsync);
   }

   // 9. Now clean up any errors that might have occurred.
   if (err != noErr) {
      gsSHInVars.recording = false;
      if (deviceOpened)
         SPBCloseDevice(gsSHInVars.inRefNum);
      if (allocated) {
         DisposeHandle(gsSHInVars.inHandle);
         gsSHInVars.inHandle = nil;
      }
   }

   return(err);
}
```

This routine, the most lengthy in Helper, is staged, and nearly every stage can fail. Each stage does its function and sets err to some error code. Subsequent stages

QUALITY OF SAMPLED SOUND

Two basic characteristics affect the quality of sampled sound: sample rate and sample size.

Sample rate, or the rate at which voltage samples are taken, determines the highest possible frequency that can be recorded. Specifically, for a given sample rate, you can sample sounds of up to *half* that frequency. For instance, if the sample rate is 22,254 samples per second (hertz, or Hz), the highest frequency you could record would be around 11,000 Hz.

A commercial compact disc is sampled at 44,100 samples per second, providing a frequency response of up to around 20,000 Hz, the limit of human hearing. Your dog, however, may find your CD player a bit wanting.

Sample size, or *quantization,* determines the dynamic range of the recording (the difference between the quietest and the loudest sound). If the sample size is eight bits, there are 256 discrete voltage levels that can be recorded. This provides approximately 48 decibels (dB) of dynamic range.

A CD's sample size is 16 bits, which provides about 96 dB of dynamic range. Humans with good hearing are sensitive to ranges greater than 100 dB, so you're likely to see 18- or 20-bit digital audio in the next ten years.

execute only if the result of the previous stage was noErr. Significant stages (like opening the sound input device and memory allocation) set flags that allow SHRecordStart to clean up if an error occurs after one of those operations.

The first stage tries to open the sound input device with SPBOpenDevice. The device's reference number is stored in the inRefNum field of the input variables record. The second stage tests the device to see if it can handle asynchronous recording. The third stage attempts to allocate the memory buffer for the recorded sound based on the parameter maxK.

The fourth stage turns on metering (which allows Helper to retrieve the instantaneous record level) and sets the recording quality based on the quality parameter (the Sound Manager recording values—'good', 'betr', or 'best'). The fifth stage retrieves device settings that Helper uses to construct the sound header. The sixth stage actually creates the header with a call to the Sound Manager routine SetupSndHeader.

The seventh stage moves the recording handle high in the heap and locks it in preparation for recording. SHRecordStart then sets up inPB, the sound input parameter block, in preparation for recording. Finally, the eighth stage flags that recording is under way, clears the application's recording-completed flag, and then initiates recording with a call to SPBRecord. If some failure occurred, the sound handle is deallocated if necessary, and the sound input device is closed if it was opened.

A note to MacApp users: You should set PermAllocation to true before calling SHRecordStart; otherwise the sound input handle may be allocated from temporary storage.•

SHGETRECORDEDSOUND

```
pascal OSErr SHGetRecordedSound(Handle *theSound)
{
   if (gsSHInVars.recordComplete) {
      if (gsSHInVars.recordErr != noErr) {
         *theSound = nil;
         return(gsSHInVars.recordErr);
      } else {
         *theSound = gsSHInVars.inHandle;
         return(noErr);
      }
   } else {
      *theSound = nil;
      return(kSHErrNoRecording);
   }
}
```

SHGetRecordedSound is used by the application to retrieve the handle of a sound that has finished recording. Once the application's recording-completed flag goes true (or SHRecordStatus indicates "finished") it's OK to call SHGetRecordedSound. If an error terminated recording, SHGetRecordedSound returns the error. If no error occurred, theSound is set as a handle to the recorded sound. The recorded sound can be played back with the Sound Manager or any of Helper's playback routines, or can be written out as a 'snd ' resource.

SHRECORDSTOP

```
pascal OSErr SHRecordStop(void)
{
   if (gsSHInVars.recording)
      return(SPBStopRecording(gsSHInVars.inRefNum));
}
```

SHRecordStop stops recording like the stop button on a tape deck. If recording was stopped before the entire input buffer was filled, SHIdle will shorten the sound handle to the correct size.

ADVANCED RECORDING ROUTINES

Three advanced routines give you more control over the recording process. SHRecordPause and SHRecordContinue pause and continue recording. SHRecordStatus returns status information about a recording sound, as well as its progress (how much has been recorded with respect to the total space that has been allocated) and the instantaneous input level.

SHRECORDPAUSE AND SHRECORDCONTINUE

```
pascal OSErr SHRecordPause(void)
{
   OSErr err;

   if (gsSHInVars.recording) {
      if (!gsSHInVars.paused) {
         err = SPBPauseRecording(gsSHInVars.inRefNum);
         gsSHInVars.paused = (err == noErr);
         return(err);
      } else return(kSHErrAlreadyPaused);
   } else return(kSHErrNotRecording);
}
```

SHRecordPause simply pauses recording with the routine SPBPauseRecording, assuming the recording is not already paused.

```
pascal OSErr SHRecordContinue(void)
{
   OSErr err;

   if (gsSHInVars.recording) {
      if (gsSHInVars.paused) {
         err = SPBResumeRecording(gsSHInVars.inRefNum);
         gsSHInVars.paused = !(err == noErr);
         return(err);
      } else return(kSHErrAlreadyContinued);
   } else return(kSHErrNotRecording);
}
```

SHRecordContinue resumes recording of a previously paused recording with the routine SPBResumeRecording.

SHRECORDSTATUS

SHRecordStatus uses an SHRecordStatusRec record to provide detailed information about the progress of a sound while it's being recorded.

```
typedef struct {
   SHRecordStat   recordStatus;      // Current recording status.
   unsigned long  totalRecordTime;   // Total (maximum) record time in ms.
   unsigned long  currentRecordTime  // Current recorded time in ms.
   short          meterLevel;        // 0..255, the current input level.
} SHRecordStatusRec;
```

```
pascal OSErr SHRecordStatus(SHRecordStatusRec *recordStatus)
{
   short          recStatus;
   OSErr          err;
   unsigned long  totalSamplesToRecord, numberOfSamplesRecorded;

   if (gsSHInVars.recording) {
      err = SPBGetRecordingStatus(gsSHInVars.inRefNum, &recStatus,
         &recordStatus->meterLevel, &totalSamplesToRecord,
         &numberOfSamplesRecorded, &recordStatus->totalRecordTime,
         &recordStatus->currentRecordTime);
      if (err == noErr)
         recordStatus->recordStatus = (gsSHInVars.paused ? shrPaused :
            shrRecording);
      else recordStatus->recordStatus = shrError;
      return(err);
   } else if (gsSHInVars.recordComplete) {
      recordStatus->recordStatus = shrFinished;
      recordStatus->meterLevel = 0;
      // Don't know about the other fields--just leave 'em.
      return(noErr);
   } else return(kSHErrNotRecording);
}
```

An SHRecordStatusRec record contains a recordStatus field that's analogous to the playback status. SHRecordStatus calls SPBGetRecordingStatus to get status information from the Sound Manager. The meter level, total record time, and current record time are placed directly in the output record.

The SHRecordStat enum looks like this:

```
typedef enum {
   shrError = -1,
   shrFinished = 0,
   shrPaused = 1,
   shrRecording = 2
} SHRecordStat;
```

The recording status is set to shrError if an error occurred on the SPBGetRecordingStatus call, shrFinished if the recordComplete flag is set, shrRecording if the sound is currently recording, or shrPaused if the sound is recording but is paused. The information in an SHRecordStatusRec, along with the other routines described in this article, is enough to support an on-screen tape deck.

USING HELPER

The best way to get a feeling for how to use Helper is to look over the source code for the small demonstration program, SHDemo, on the CD. It demonstrates triggered sounds using SHPlayByID; continuous background music using SHPlayByHandle and SHGetChannel; and a mini tape deck with a level meter, progress bar, and record, stop, play, and pause buttons that work for both recording and playback. SHDemo exercises all of Helper's calls, so you're likely to find appropriate examples somewhere inside SHDemo. For a practical example of what Helper can do, take a look at the RapMaster application on the CD.

JOIN THE NOISY REVOLUTION

Consider how sound, as a data type, might fit into and enhance your application. You'll still need to implement the user interface, but Helper can shield you from many of the ugly Sound Manager details described above, and can also form the basis for a customized sound package better suited to the specific needs of your application. Either way, join the Noisy Revolution today!

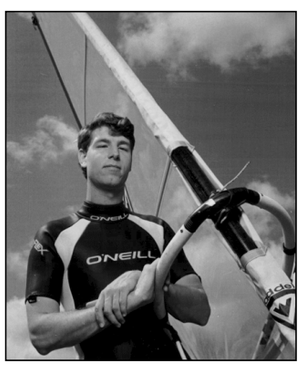
RELATED READING

- *Inside Macintosh* Volume VI (Addison-Wesley, 1991), Chapter 22, provides comprehensive information on the latest version of the Sound Manager, including information on sound input.
- *Inside Macintosh* Volume V (Addison-Wesley, 1988), Chapter 2, provides user interface guidelines for the inclusion of sound in Macintosh applications.
- *Inside Macintosh* Volume II (Addison-Wesley, 1985), Chapter 8, and Volume V, Chapter 27, provide a historical perspective on sound on the Macintosh, if you're curious. The information in these chapters is superseded by Volume VI, Chapter 22.

THANKS TO OUR TECHNICAL REVIEWERS
Rich Collyer, Neil Day, Kip Olson, Jim Reekes•

AROUND AND AROUND: MULTI-BUFFERING SOUNDS

The main problem with digital audio is that the data often exceeds the amount of available memory, forcing programmers to resort to multiple-buffering schemes. This article presents one such technique, in the form of a program called MultiBuffer, and explores some interesting things you can do along the way.



NEIL DAY

When dealing with digital audio, you're frequently going to find yourself in situations where the sample you want to play won't fit in the memory you have available. This leaves you with several alternatives: you can play shorter sounds; you can try to squeeze the sound down to a more manageable size by resampling it at a lower frequency or by compressing it (both of which will degrade the fidelity of the sound); or you can try to fool the machine into thinking it has the whole sample at its disposal. In cases where you don't want to compromise length or quality, trickery is your only option.

If you've spent any time with the Sound Manager, you no doubt have run across the routine SndPlayDoubleBuffer, which provides one reasonably straightforward method of implementing a double-buffering scheme. The advantage of using SndPlayDoubleBuffer is that it allows you to get a fairly customized double-buffering solution up and running with very little work. You need only write a priming routine for setting up the buffers and filling them initially, a DoubleBack procedure that takes care of swapping buffers and setting flags, and a read service routine for filling exhausted buffers; the Sound Manager handles all the other details. SndPlayDoubleBuffer is in fact used by the Sound Manager's own play-from-disk routine, SndStartFilePlay.

If your program will simply play from disk, your best bet is probably either SndPlayDoubleBuffer or SndStartFilePlay. Both offer good performance painlessly, saving you development time and avoiding the need to understand the Sound Manager to any great degree. If, however, you want to do some snazzier things with your sound support, such as adding effects processing, a deeper understanding of multiple buffering is essential. Read on . . .

NEIL DAY When Neil isn't glued to his Macintosh, working on one of his various programmatic whatnots, you can usually find him strapped to a piece of sporting equipment leaping off something. Neil's favorite jumping-off points are waves, cliffs, and cornices, in that order.•

PROCESSING SOUNDS WITH THE ASC

Audio support on the Macintosh computer is handled by the Apple Sound Chip (ASC), which takes care of converting the digital representation of your sound back to analog, which can then be played by a speaker attached to your Macintosh. (See "Sound: From Physical to Digital and Back" for a description of this process.)

You can think of the ASC as a digital-to-analog converter with two 1K buffers to hold the data to be processed. When either of the buffers reaches the half-full mark, the ASC generates an interrupt to let the Sound Manager know that it's about to run out of data. Because of this, it's important to make sure that your buffers are a multiple of 512 bytes, since in an attempt to keep the ASC happy the Sound Manager will pad your data with silence if you choose an "odd" buffer size. In the worst case this can lead to annoying and mysterious silences between buffers, and at best it will hurt your performance. This doesn't mean that you need to limit yourself to 512- or 1024-byte buffers: The Sound Manager takes care of feeding large buffers to the ASC a chunk at a time so that you don't have to worry about it. As long as your sound is small enough to fit into available memory, you can play it simply by passing the Sound Manager a pointer to the buffer containing the sample.

Assuming that the ASC's buffers never run dry, it will produce what seems to be a continuous sound. As long as you can keep handing it data at a rate greater than or equal to the speed at which it can process the data, there won't be any gaps in the playback. Even the best-quality samples, like those found on audio CDs, play back at the leisurely rate of 44,100 sample frames per second (a frame consists of two sample points, one for each channel of sound), a rate that the processors of 68020-based Macintosh computers and SCSI devices can keep up with. All you need to do is hand one buffer to the Sound Manager to play while you're filling another. When the buffer that's currently playing is exhausted, you pass the recently filled one to the Sound Manager and refill the empty one. This process is the digital equivalent of the venerable bucket brigade technique for fighting fires.

CONTINUOUS SOUND MANAGEMENT

This section discusses a general strategy for actually making a multibuffering scheme work. First, however, I want to touch on some of the properties and features of the Sound Manager that we'll exploit to accomplish multibuffering. If you're already familiar with the Sound Manager, you may want to skip ahead to the section "When You're Done, Ask for More."

CHANNELS, QUEUES, COMMANDS, AND CALLBACKS

The atomic entity in the Sound Manager is a channel. A channel is essentially a command queue linked to a synthesizer. As a programmer, you issue commands to the channel through the Sound Manager functions SndDoCommand and SndDoImmediate. The Sound Manager executes the commands asynchronously,

SOUND: FROM PHYSICAL TO DIGITAL AND BACK

What you experience when you hear a sound is actually your ear picking up a series of pressure changes, commonly thought of as waves, in the ambient air. Through a bunch of physiological magic, the ear converts these pressure changes to a neural signal that your brain can then recognize as your alarm clock or Chopin, depending on the waveform. Both the waveform and its neural equivalent are analog, which is useless as far as your computer is concerned. To get the analog phenomena into your machine, you need to use some sort of transducer (a microphone, for instance) and an analog-to-digital converter such as the sound input hardware found in most Macintosh models.

Like your ear, the microphone picks up minute pressure changes in the air, but it produces an electrical signal that the analog-to-digital converter turns into a stream of numbers corresponding to the voltage (amplitude) of the signal. Each of these numbers is a discrete *sample point*, and the collection of sample points that define the waveform are collectively known as a *sample*.

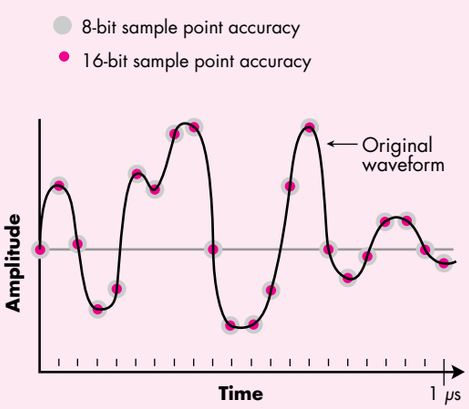
In an ideal world your sample would be continuous, meaning that there would be an infinite number of sample points for any given time period. The reality is that analog-to-digital (and digital-to-analog) converters can handle only a finite number of sample points per second, so the concept of *sample frequency* becomes important. The higher the frequency, the better the sample approximates the original waveform. The sample frequency is usually expressed in kilohertz, which gives the number of sample points per microsecond. Common sample rates are 7 kHz, 11 kHz, and 22 kHz, though the Sound Manager currently supports any sample rates between 1 kHz and 22 kHz. In practice, it's best to stick to an even divisor of 22 kHz, since it minimizes the number of hoops the software needs to jump through to play your sound.

Another factor that affects the fidelity of the sample is its *quantization*, which is related to how many bits are used to describe each sample point. The Sound Manager currently supports only 8-bit samples, which is sufficient for most applications.

The digitized sound is stored either in memory or on a more permanent storage medium such as a hard disk or CD-ROM. The important thing to take away from this discussion is that in order to get high-quality samples, you need to have a high sampling rate and a reasonable sample size (read: lots of memory).

Converting samples back into sound is exactly the reverse of the recording process. The data goes to a digital-to-analog converter, which generates a specific voltage based on the digital value handed to it. After some sort of amplification, these voltages cause a speaker to emit an "image" of the originally sampled waveform.

The following figure shows a 1-microsecond snapshot of a waveform sampled at a rate of 22 kHz and digitized at sample sizes of 8 and 16 bits. The dots show how much the digitized waveform can vary from the original waveform at each sample size. The 16-bit size gives a better approximation of the original waveform, but eats up a lot of memory.



returning control to the caller so that your application can continue with its work. The difference between the two functions is that SndDoCommand will always add your request to a queue, whereas SndDoImmediate bypasses the queuing mechanism and executes the request immediately. It's important to understand that at the lowest level the Sound Manager always executes asynchronously—your program regains control immediately, whether the call is queued or not.

We're interested here in two sound commands, bufferCmd and callBackCmd.

- bufferCmd sends a buffer off to the Sound Manager to be played. The buffer contains not only the sample, but also a header that describes the characteristics of the sound, such as the length of the sample, the rate at which it was sampled, and so on.
- callBackCmd causes the Sound Manager to execute a user-defined routine. You specify this routine when you initialize the sound channel with the Sound Manager function SndNewChannel. Be aware that the routine you specify executes at interrupt time, so your application's globals won't be intact, and the rules regarding what you can and can't do at interrupt time definitely apply.

WHEN YOU'RE DONE, ASK FOR MORE

The key to achieving continuous playback of your samples is always to have data available to the ASC. To keep the ASC happily fed with data, your code needs to know when the current buffer is exhausted, so that it can be there to hand over another buffer. Most asynchronous I/O systems provide completion routines that notify the application when an event terminates. Unfortunately, such a routine is not included in the current incarnation of the output portion of the Sound Manager. In the absence of a completion routine, the best way to accomplish this type of notification is to queue a callBackCmd immediately following a bufferCmd. For the purpose of this discussion, the bufferCmd-callBackCmd pair can be considered a unit and will be referred to as a *frame* from here on. Since it's often not practical to play an entire sample in one frame, you'll probably need to break it up into smaller pieces and pass it to the Sound Manager a frame at a time. Figure 1 illustrates how a sample too large to fit in memory is broken up into frames consisting of a bufferCmd and callBackCmd.

To further reinforce the illusion of a frame being a standalone entity, it's useful to encapsulate the bufferCmd-callBackCmd pair in a routine. A bare-bones version of a QueueFrame routine might look like this:

```
OSErr QueueFrame (SndChannelPtr chan, Ptr sndData)
{
   OSErr       err;
   SndCommand  command;
```

```
    command.cmd = bufferCmd;
    command.param1 = nil;
    command.param2 = (long) sndData;
    err = SndDoCommand (chan, &command, false);
    if (err)
       return err;

    command.cmd = callBackCmd;
    command.param1 = nil;
    command.param2 = nil;
    err = SndDoCommand (chan, &command, false);
    if (err)
       return err;
}
```

By queuing up another frame from the callback procedure, you can start a chain reaction that will keep the sample playing. Be sure to have another frame ready and waiting before the Sound Manager finishes playing the current frame. Failure to do this will cause a gap in the playback of the sound, often referred to as *latency*.

Two important factors that can cause latency are the speed of your source of sound data and the total size of the buffers you're using. The faster your data source, the smaller the buffer size you can get away with, while slower sources require larger buffers. For example, if you're reading from a SCSI device with an average access time of 15 milliseconds, you can keep continuous playback going with a total of about 10K of buffer space; if your source is LocalTalk, plan on using significantly larger buffers. You may need to experiment to find the optimal buffer size.

A third factor that can contribute to latency is the speed at which your callback code executes. It's very important to do as little work as possible within this routine, and in extreme cases it may be advantageous to write this segment of your code in assembly language. Of course, faster 68000-family machines will let you get away with more processing; a routine that may require hand coding to run on a Macintosh Plus can probably be whipped off in quick-and-dirty C on a Macintosh Quadra. As is the case with all time-critical code on the Macintosh, it's important to take into account all the platforms your code may run on.

Once you've compensated for any potential latency problems, this method of chaining completion routines has a couple of advantages:

- After you start the process by queuing the first frame, the "reaction" is self-sustaining; it will terminate either when you kill it or when it runs out of data.
- Once started, the process is fully asynchronous. This gives your application the ability to continue with other tasks.

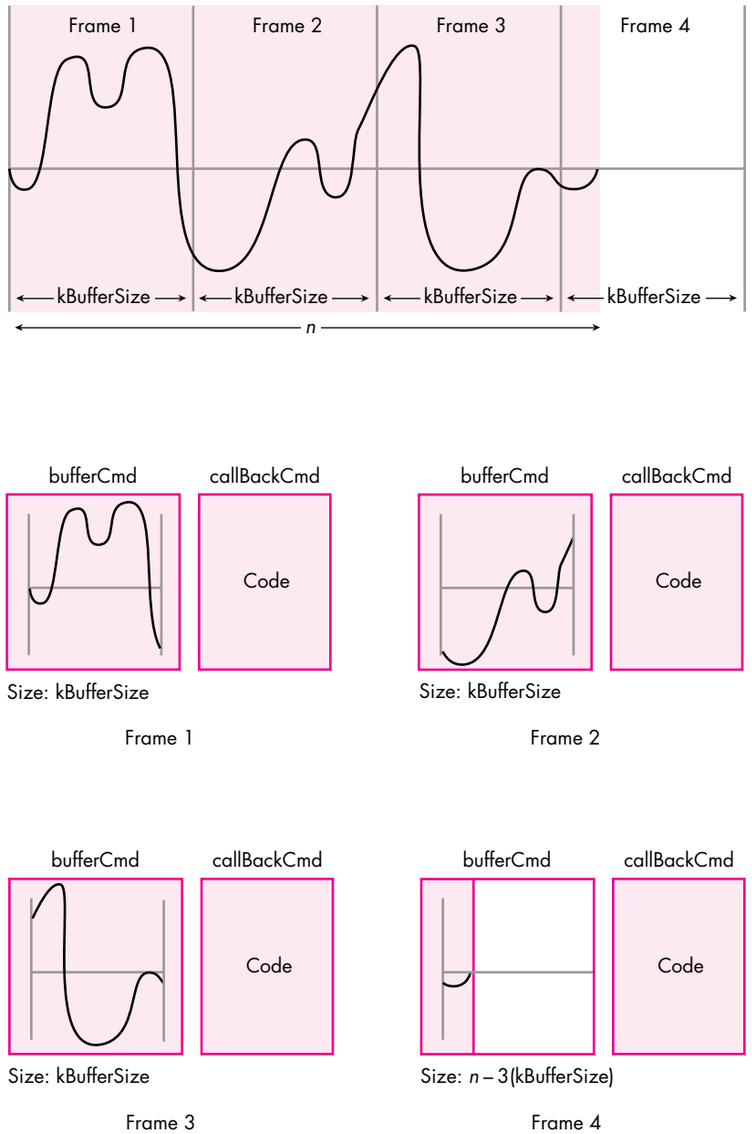


Figure 1
Dividing a Sample into Frames

Your callback procedure must take care of three major functions. It must queue the next frame, refill the buffer that was just exhausted, and update the buffer indices. In pseudocode, the procedure is as follows:

```
CallbackService ()
{
   //
   //    Place the next full buffer in the queue.
   //
   QueueFrame (ourChannel, fullBuffer);
   //
   //    Refill the buffer we just finished playing.
   //
   GetMoreData (emptyBuffer);
   //
   //    Figure out what the next set of buffers will be.
   //
   SwapBuffers (fullBuffer, emptyBuffer);
}
```

WHAT MAKES MULTIBUFFER DIFFERENT?

The previous section discussed general tactics for multiple-buffering models and for chaining callback routines; these would be used by any continuous play module. MultiBuffer, included on the *Developer CD Series* disc, uses these basic concepts as its foundation, but differs in several important ways in order to address some performance issues and attain a higher level of flexibility.

Thus far the discussion has centered on *playback-driven* buffering models, in which the completion routine is keyed to the playback. This model, on which MultiBuffer is based, is appropriate for applications that play from a storage device or that play from synthesis. Playing from a real-time source, such as a sound input device or a data stream coming over a network, requires a *source-driven* buffering model, in which the callback is associated with the read routine. There's little difference between these two models, but using the wrong model can lead to the loss of small amounts of sound data.

The major design goal for MultiBuffer was to make it modular enough to be easily customized. It includes an independent procedure for reading data from the source (ReadProc), as well as a procedure for processing the raw data obtained from the ReadProc (ProcessingProc). MultiBuffer also allows you to work with more than two buffers; simply modify the constant kNumBuffers. In some situations, more than two buffers can be handy, such as instances where you want to reduce the lag between playback time and real time. Several classes of filter require that you have a fairly extensive set of data available for processing. Pulse-response filters, low- and high-

pass (first-order) filters, and spectral-compression filters are all examples of applications in which multiple buffers can simplify implementation. It's important to realize, however, that using many buffers introduces extra overhead, so your buffer sizes will need to be correspondingly larger. Because of this added overhead, you can end up in a Catch-22 situation; there is a point at which the benefit of having more buffers is negated by the increase in buffer size.

The optional processing procedure allows you to perform some simple modifications on the data before it's played. It's vital that you keep the issue of latency in mind when dealing with your processing procedure; it can have a profound effect on the amount of time required to ready a buffer for play. Since this is a time-critical section of the buffering code, it's often desirable to write this procedure in assembly language to squeeze out the highest performance possible.

Because the procedures for reading sound data and for processing the data are separate modules, MultiBuffer is quite flexible. The program includes a simple example of this flexibility. One of the playback options is to play an Audio Interchange File Format (AIFF) file backward. To achieve this, I altered ReadProc to read chunks of the target file from end to beginning, then used ProcessingProc to reverse the contents of the buffer. If you take a look in PlayFromFile.c, you'll find that the differences between the functions ReadProc and ProcessingProc and their counterparts BackReadProc and BackProcessingProc are minimal.

HOW IT HANGS TOGETHER

MultiBuffer is basically a series of three sequentially chained interrupt routines. These are the callBackProc, a read completion routine, and a deferred task that takes care of any processing that needs to be done on the raw data. Deferred tasks are essentially interrupt routines that get put off until the rest of the regularly scheduled interrupts have completed. A deferred task won't "starve" important interrupts executing at a lower priority level. Such tasks also have the advantage of executing when interrupts have been reenabled, allowing regular interrupt processing to continue during their execution. Unfortunately, a deferred task is still bound by the restrictions on moving or purging memory placed on regular interrupt routines.

The routine DoubleBuffer begins the execution. It takes care of setting up private variables, priming the buffers, and initiating the play that starts the chain reaction.

MultiBuffer is composed of six main files, each of which deals with one of the functional aspects of MultiBuffer.

- MainApp.c contains the application framework for the MultiBuffer demo.
- DoubleBuffers.c includes all the code for dealing with multibuffering. For most applications, you shouldn't need to modify any of the code in this file.

- AIFFGoodies.c features fun with parsing AIFF header information. The basic purpose of the code in this file is to get pertinent information out of an AIFF file and into a sound header.
- PlayFromFile.c contains the routines for playing an AIFF file forward and backward.
- PlayFromSynth.c has the code necessary for playing a wave segment as a continuous sound.
- Oscillator.c is responsible for generating one cycle of a sine wave at a specified frequency and amplitude.

NUTS AND BOLTS

The rest of this article goes into gory detail on the inner workings of MultiBuffer. You'll find this helpful if you plan on modifying the code to suit your own needs, or if you want to gain a painfully in-depth understanding of the processes involved.

CONSTANTS

There are only two constants you need to worry about.

kBufferSize. This indicates the size of the buffer in bytes. The larger this value, the more time you have between buffer switches. It should be a multiple of 512.

kNumBuffers. The value of this constant determines the number of buffers that MultiBuffer uses. In most cases, this value should be 2.

IMPORTANT DATA STRUCTURES

Listed here are some of the important data structures used by MultiBuffer. All of them can be found in the file DoubleBuffer.h.

```
typedef struct {
   ParamBlockHeader
   short       ioFRefNum;
   long        filler1;
   short       filler2;
   Ptr         ioBuffer;
   long        ioReqCount;
   long        ioActCount;
   short       ioPosMode;
   long        ioPosOffset;
} strippedDownReadPB, *strippedDownReadPBPtr;
```

The strippedDownReadPB structure is the minimal parameter block the Device Manager needs in order to execute a read from a file. A primary concern was keeping

MultiBuffer's overhead very low, and since each buffer needs to have a parameter block associated with it, using the full-blown ParamBlockRec was undesirable.

```
typedef struct {
   strippedDownReadPB   pb;
   Ptr                  userInfo;
   short                headerNum;
} ExtParamBlockRec, *ExtParmBlkPtr;
```

ExtParamBlockRec is a wrapper that adds a few pieces of MultiBuffer-specific information to the end of a parameter block. It allows us to get information about the state of the program to the completion routine without using any globals, which would make for ugly code.

```
typedef struct {
   short                flags;
   ExtParamBlockRec     readPB;
   DeferredTask         dt;
   SoundHeaderPtr       header;
} SampleBuffer, *SampleBufferPtr;
```

SampleBuffer contains all the information necessary for managing samples. The flags field holds the current status of the buffer. The readPB field is the parameter block the Device Manager uses to read the data from the source. Since it's possible that you'll have more than one asynchronous read queued at a time, reusing parameter blocks is inadvisable, hence the need for one associated with each buffer. The dt field is a deferred task record that will be used to install the ProcessingProc. The header field is a pointer to a sound header that the Sound Manager uses to play a sample. Note that the sound header definition, found in Sound.h, contains a samplePtr that will have a nonrelocatable block associated with it for holding the actual sample data.

```
typedef struct {
   OSType               signature;
   long                 refNum;
   long                 fileDataStart;
   long                 bytesToGo;
   short                currentBuffer;
   SampleBuffer         buffers[kNumBuffers];
   SndCommand           bCmd;
   SndCommand           cbCmd;
   SndCallBackProcPtr   oldCallBack;
   long                 oldUserInfo;
   long                 a5ref;
} PrivateDBInfo, *PrivateDBInfoPtr;
```

PrivateDBInfo is a private data structure that contains all the information MultiBuffer requires to do its thing. The refNum field contains the reference number of the file or device that contains the sound data we're going to play. As implemented here, MultiBuffer reads information from a file on an HFS device, so refNum will contain a File Manager file reference number. It's also possible to use a sound input device or the network as the source for your sound data. In such cases refNum would contain a sound input device reference number or an AppleTalk unit reference number, respectively. The field fileDataStart contains the position in the file at which the actual sound data starts; since AIFF files and resources can contain tens of bytes of header information, it's important to be able to locate the start of our data. The bytesToGo field keeps track of the number of bytes of sound data to be played. This field may not be meaningful in the case of continuous sound sources, since the data stream doesn't necessarily have a definite end.

The currentBuffer field contains the number of the buffer that's currently playing. This field can actually be thought of as an index into the array of SampleBuffers. In this array, buffers[kNumBuffers] contains the information needed for actually playing the sound through the Sound Manager (and some other convenient goodies, too).

bCmd and cbCmd are sound commands that are used to send a frame off to be played by the Sound Manager. bCmd contains the information necessary to issue a bufferCmd, and cbCmd is used to issue a callBackCmd. Both of these structures are used frequently with little modification; by having them preinitialized and easily accessible to the routines that need them, we save a few instructions.

oldCallBack and oldUserInfo are holding spots for any userInfo and callBack data stored in the SndChannel data structure before MultiBuffer was called. MultiBuffer places its own information in the userInfo and callBack fields, so it's important to save and restore any values that may have been lurking there previously.

a5Ref contains a reference to the application's A5 world, so that we can access the global variables that MultiBuffer uses at times when A5 may be invalid, such as at interrupt time.

THE ROUTINES

This section describes support routines that make up MultiBuffer, many of which can be used unchanged in your own code.

```
short OpenAIFFFile (void)
```

Found in: AIFFGoodies.c

OpenAIFFFile is a pretty generic Standard File-based routine that filters out all file types other than 'AIFF' files. After the user selects a file, this routine opens it and returns the file reference number or an OSErr.

```
long GetAIFFHeaderInfo (short frefNum, SoundHeaderPtr theHeader)
```

Found in: AIFFGoodies.c

GetAIFFHeaderInfo is an example of how to parse the header information out of an AIFF file. The basic strategy is to read pieces of the header into a buffer, where a struct template can be overlaid onto the data, making the values easy to access. Important information from the AIFF file can then be put into a SoundHeader data structure, which will be used later when data is passed to the Sound Manager for playing. The routine provided in MultiBuffer extracts only the sound data parameters, such as the length, sample rate, sample size, and number of channels. Other chunks are currently ignored, although the code is there to support siphoning the information out of them. GetAIFFHeaderInfo leaves the file mark at the beginning of the sound data and returns the number of bytes of sound data in the file.

One limitation of the current implementation of this routine is that it deals only with 8-bit monophonic sounds.

```
OSErr RecordAIFFFile (OSType creator)
```

Found in: AIFFGoodies.c

RecordAIFFFile uses the sound input routine SndRecordToFile to record a sound to an AIFF file. In preparation for this, it uses the Standard File Package to select a file to record to and opens the file, creating it if it doesn't exist, clearing it if it does. One of the great features of the sound input portion of the Sound Manager is that it provides routines with standard user interfaces for recording. As this routine illustrates, all you need to worry about is passing a valid file reference number and quality selector to SndRecordToFile; the rest is taken care of.

```
OSErr DoubleBuffer (SndChannelPtr chan, unsigned long fileRefNum, ProcPtr
         readproc, ProcPtr processproc, SoundHeaderPtr generalHeader,
         unsigned long playSize, long dataOffset, Ptr *privateData)
```

Found in: DoubleBuffers.c

DoubleBuffer is the main application interface to the buffering routines. It takes care of all the setup required as well as initiating the buffering "chain reaction." In the spirit of a picture being worth a thousand words, the routine follows.

```
OSErr DoubleBuffer (SndChannelPtr chan, unsigned long fileRefNum, ProcPtr
         readproc, ProcPtr processproc, SoundHeaderPtr generalHeader,
         unsigned long playSize, long dataOffset)
{
   OSErr               err      = noErr;       // error bucket
   PrivateDBInfoPtr    dbInfo   = nil;
```

```
// Clear the global stop flag.
gsStopFlag = false;

// We're going to use a PrivateDBInfo structure to hold all the
// information we'll need later on to do our double buffering. The
// next several lines of code deal with allocating space for it and
// its members and initializing fields.
dbInfo = SetUpDBPrivateMem ();
if (dbInfo != nil) {
   DebugMessage ("\p Allocated dbInfo successfully");

   // Return a pointer to MultiBuffer's private data structure so the
   // caller can dispose of it when the operation finishes.
   *privateData = (Ptr)dbInfo;

   // Install the read procedure. This is mandatory.
   if (readproc == nil) {
      Assert (readproc == nil, "\pNo readproc specified");
      FreeDBPrivateMem (dbInfo);             // say bye to memory
   } else {
      DebugMessage ("\p Have a valid readproc");
      dbInfo->readProcPtr = readproc;

      // Install the processing procedure (if any). Lack of a
      // processing procedure knocks one level of interrupt processing
      // out, so this is a good way to save time and decrease the
      // minimum buffer size.
      dbInfo->processingProcPtr = processproc;

      dbInfo->refNum = fileRefNum;           // store file ref num
      dbInfo->a5ref = SetCurrentA5 ();

      // We're essentially going to take over the specified sound
      // channel to do double buffering with it; as a result, we'll
      // install our own callback and put private data structures in
      // the userInfo field. We're going to save the values that were
      // there when we started, in case they shouldn't be stomped on.
      if (chan->userInfo)                       // valid userInfo?
         dbInfo->oldUserInfo = chan->userInfo;  // save it
      chan->userInfo = (long) dbInfo;           // pointer to our vars

      DebugMessage ("\pAbout to prime buffers");
      dbInfo->bytesToGo = playSize;         // set up play size
      dbInfo->fileDataStart = dataOffset;   // offset into data stream
      err = PrimeBuffers (dbInfo, generalHeader);  // fill buffers
```

```
        if (err != noErr) {
           // If we got to here, we got one [censored] of an error
           // trying to read the buffers, so now we commit programmatic
           // seppuku.
           Assert (err != noErr,
                    "\pHit an error trying to Fill buffers");
           FreeDBPrivateMem (dbInfo);            // say bye to memory
        } else {
           DebugMessage ("\pSuccessfully primed buffers");
           // Presumably at least one of our buffers has been filled,
           // so let's set the chain reaction in motion. Note that
           // there is a possibility that we got only one buffer half
           // full. No worries: the callback routine will handle that
           // nicely!!

           dbInfo->bCmd.cmd = bufferCmd;
           dbInfo->bCmd.param1 = nil;
           dbInfo->bCmd.param2 = (long) dbInfo->buffers[0].header;

           dbInfo->cbCmd.cmd = callBackCmd;
           dbInfo->cbCmd.param1 = nil;
           dbInfo->cbCmd.param2 = nil;

           err = QueueFrame (chan, dbInfo);
           DebugMessage ("\pJust finished queueing up the first frame");
        }
     }
  }
  return (err);
}
```

```
OSErr QueueFrame (SndChannelPtr chan, PrivateDBInfoPtr dbInfo)
```

Found in: DoubleBuffers.c

QueueFrame takes care of passing a bufferCmd containing the sound data to be played followed by a callBackCmd to the Sound Manager, making sure that the data is valid. QueueFrame also updates the pointer to the next buffer to be played.

```
pascal void DBService (SndChannelPtr chan, SndCommand* acmd)
```

Found in: DoubleBuffers.c

When the Sound Manager receives a callBackCmd indicating that a buffer has finished playing, DBService queues up the next buffer in line to be played and calls the user-specified ReadProc to refill the exhausted buffer.

```
void CompleteRead (void)
```

Found in: DoubleBuffers.c

Upon completion of the asynchronous read queued by ReadProc, CompleteRead is called to handle errors and queue up a deferred task to perform any processing necessary on the freshly read data. Note that this routine uses the inline functions getErr and getPB to retrieve the error value from register D0 and a pointer to the parameter block from register A0, respectively.

```
OSErr ReadProc (void *private, short bufNum, Boolean asynch)
OSErr BackReadProc (void *private, short bufNum, Boolean asynch)
```

Found in: PlayFromFile.c

ReadProc is one of the few routines you may want to modify for your specific application. It takes care of reading data from the input source—in this case an AIFF file on a hard disk.

The three versions provided in the MultiBuffer application are ReadProc, BackReadProc, and WaveReadProc. ReadProc reads data starting at the mark and moving forward, BackReadProc starts at the end of the data and reads toward the start, and WaveReadProc fakes a continuous stream of data from a wave snippet by filling the buffer with copies. By replacing ReadProc, you can easily customize the behavior of your application.

```
void ProcessingProc (void)
void BackProcessingProc (void)
```

Found in: PlayFromFile.c

The ProcessingProc routine does any processing needed on the freshly read buffer. The amount of time you can spend in this routine depends directly on the size of your buffers. This is one of the key areas in which latency problems can occur.

In the MultiBuffer code, ProcessingProc simply converts from 2's complement notation to binary offset notation. BackProcessingProc reverses the buffer as well as converts it. AIFF files are by definition in 2's complement notation, whereas the Sound Manager understands only binary offset notation, making this conversion necessary. Binary offset notation is a somewhat peculiar format; its zero point is at $80. $FF corresponds to the maximum amplitude and $0 is the minimum amplitude of a wave.

If you're planning on doing any processing on your data, it's strongly recommended that you write the code in assembly language, since your code will likely execute far faster.

```
OSErr PrimeBuffers (PrivateDBInfoPtr dbInfo)
```

Found in: DoubleBuffers.c

PrimeBuffers fills each of the allocated buffers with data. This gives the buffering routines some data to work with on the first trip through the buffering cycle.

An interesting aspect of this routine is that it uses ReadProc to get the data from the source and ProcessingProc to transform it to the desired state.

```
PrivateDBInfoPtr SetUpDBPrivateMem (void)
```

Found in: DoubleBuffers.c

SetUpDBPrivateMem allocates memory for the PrivateDBInfo data structure and initializes its fields.

```
void FreeDBPrivateMem (void *freeSpace)
```

Found in: DoubleBuffers.c

FreeDBPrivateMem releases all the memory allocated by SetUpDBPrivateMem. Zowee.

```
pascal long getPB ()
```

Found in: DoubleBuffer.h

When a completion routine is called, register A0 will contain a pointer to the parameter block of the caller. Since MPW C doesn't have support for directly accessing registers, the inline function getPB moves register A0 onto the stack, where the C compiler can figure out how to assign it to a variable.

```
pascal short getErr ()
```

Found in: DoubleBuffer.h

The getErr function does the same thing as getPB, except that it deals with the error code (found in register D0) instead of the parameter block pointer.

```
pascal SampleBufferPtr getDTParam ()
```

Found in: DoubleBuffer.h

The Deferred Task Manager places an optional argument to its service routine in register A1. As was true with getPB and getErr, we need to use an inline assembly function, getDTParam, to retrieve the argument.

```
pascal void CallDTWithParam (ProcPtr routine, SampleBufferPtr arg)
```

Found in: DoubleBuffer.h

So that we don't need to have two copies of ProcessingProc present, the inline function CallDTWithParam allows you to call a deferred task service routine with an argument directly. It takes the argument passed to it and puts it in register A1, then JSRs to the service routine.

```
pascal void QuickDTInstall (DeferredTaskPtr taskEl)
```

Found in: DoubleBuffer.h

The QuickDTInstall procedure saves us a few cycles by bypassing the trap dispatcher when we install a deferred task service routine. The address of the service routine is loaded into A0, then we jump directly to the installation procedure pointed to by the low-memory global jDTInstall.

This is one of the few legitimate reasons to access a low-memory global, but it can potentially get you in trouble. It works fine under system software through version 7.0.1, but is certainly a future compatibility risk. The alternative is to call DTInstall in the normal manner, but even the few milliseconds you spend in the trap dispatcher will have an adverse effect on the amount of processing you can do on your sounds.

```
pascal void Reverse (Ptr buffer, long length)
```

Found in: PlayFromFile.c

The Reverse routine does the real processing on the freshly read buffer. It reverses the buffer passed to it as well as converting it from 2's complement notation to binary offset format.

```
Ptr NewWaveForm (unsigned char amplitude, unsigned short frequency)
```

Found in: Oscillator.c

NewWaveForm generates a waveform based on the values most recently read from the controls in the application window. Amplitude must be a value between 0 and $80, while frequency can be anything within reason. This routine returns one cycle of the waveform requested.

TIPTOE THROUGH THE INTERRUPTS

As you've probably gathered from the descriptions of the routines, MultiBuffer is essentially a maze of self-perpetuating interrupt routines. This makes keeping track of what's happening at any given moment a real pain, not to mention that it severely complicates debugging. In the interest of sparing you a headache or two trying to figure out the flow of processing, let's walk through a bit of MultiBuffer's execution.

Initialization. This particular phase of execution happens at normal run time and is fairly uninteresting. The process is as follows:

1. SetUpDBPrivateMem gets called to allocate our private memory.
2. Fields of the PrivateDBInfo structure are initialized. This step includes saving the current A5 world, putting pointers to ReadProc and ProcessingProc into the appropriate fields, and saving copies of the sound channel's callBack and userInfo values.
3. PrimeBuffers is called to prefill each of the buffers with sound data.

The chain reaction. The last thing that happens in the DoubleBuffer routine is a call to QueueFrame, which places a bufferCmd followed by a callBackCmd in the channel's queue. Since both these operations will be executed asynchronously, control returns immediately to DoubleBuffer and subsequently to your application.

Figure 2 illustrates how things unfold from here. The buffer flags are set to kBufferPlaying, to indicate that the buffer is busy. As soon as the first buffer is exhausted, its associated callBackCmd causes an interrupt that transfers control to DBService, where the next frame is queued, assuming its flags indicate readiness (kBufferReady). To keep things going smoothly, a read is issued to refill the recently exhausted buffer, and its flags are set to kBufferFilling. At this point, control returns to whatever process was going on when the callBackCmd generated the interrupt.

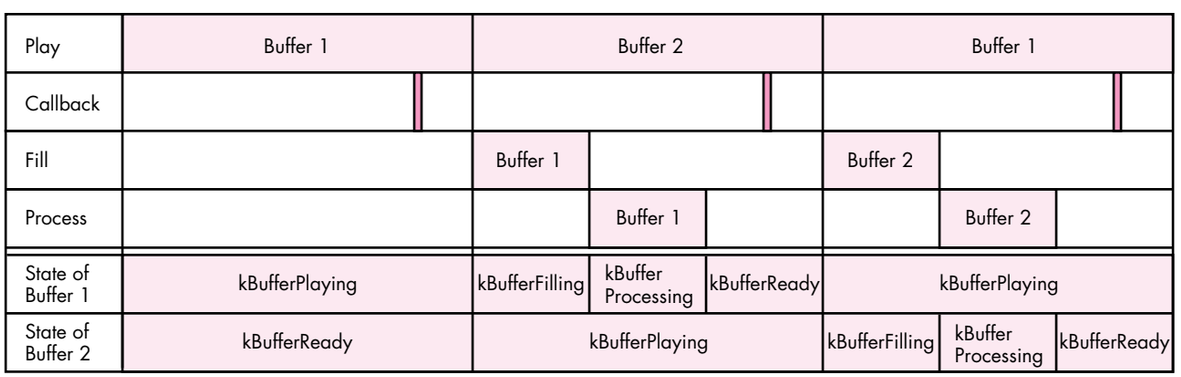


Figure 2
The Illustrated Execution, or What Happens When

The next phase starts as soon as the read queued in DBService completes, transferring control (again, at interrupt time) to the completion routine CompleteRead. If a processing procedure has been installed, a deferred task is

initiated for the buffer, and its flags are set to kBufferProcessing. Upon completion of the processing procedure, the buffer's flags are reset to kBufferReady and control returns to the main stream of execution.

The importance of using a deferred task may not be obvious, since it would seem to make sense just to call ProcessingProc at the end of CompleteRead. Doing so, however, would cause the program to spend too much time in the interrupt service routine, shutting out other critical functions being performed at a lower interrupt priority. Using a deferred task also means that the processing procedure executes after interrupts have been reenabled, which allows us to spend a little more time processing without causing the system to grind to a halt and die. Remember, though, that the total amount of time required to read in a new chunk of data and process it cannot exceed the time it takes to play a buffer, or you'll run into latency problems.

THE CUSTOM SHOP

Customizing MultiBuffer simply involves replacing ReadProc and ProcessingProc with your own routines.

ReadProc should take care of reading data from some source. It needs to fill the buffer indicated by the argument bufNum with sound data. It should have the ability to behave synchronously or asynchronously as indicated by the asynch argument. Remember to specify CompleteRead as the completion routine; otherwise MultiBuffer won't work. Depending on the device that you're reading from, you may have to use a different type of parameter block. strippedDownReadPB is a minimal parameter block for use with the File Manager; AppleTalk and the input portion of the Sound Manager will both require substituting a different parameter block for the strippedDownReadPB. All you need to modify is the definition of the data structure ExtParamBlockRec in DoubleBuffer.h. ReadProc often executes at interrupt time, so the prohibitions against anything like moving or purging memory apply.

You absolutely must have a ReadProc. The ProcessingProc, on the other hand, is optional. If specified, this routine allows you to do some limited processing on the data read from the source before it goes off to the Sound Manager to be played. Since this is a deferred task service routine, the argument is placed in A1. MultiBuffer passes this routine a pointer to the SampleBuffer to be processed. You can do anything you want to this buffer, as long as it completes relatively quickly. Remember, this is the routine that's often responsible for causing latency problems.

While this doesn't necessarily require writing additional code, DoubleBuffer expects a generic sound header for the data you're interested in playing. The only information you really need is the sample rate and the base note, since MultiBuffer sets up the other fields in the sound header. In the case of AIFF files, this requires parsing the header information in the file; for the wave-play operation, fudging a header with constants that describe the type of wave is acceptable.

IMPLEMENTATION EXAMPLES

To show how you might customize MultiBuffer, I've provided a couple of routines that make use of the package. Both have examples of how to implement a ReadProc, and the play-from-file example also takes advantage of a ProcessingProc.

PlayFromFile. PlayFromFile.c contains routines that play an AIFF file. These are good examples of how you can use alternate ReadProcs and ProcessingProcs to customize the output of your data.

PlayFromSynth. PlayFromSynth.c and Oscillator.c contain the routines necessary to implement a basic sine wave synthesizer. The only really notable feature of this implementation is the use of the refNum field of the PrivateDBInfo structure as a pointer to a single wave cycle. My thinking here was that refNum really points to the data source, unlike a traditional refNum, so why not use it as such?

FUN THINGS TO DO WITH MULTIBUFFER

Here are a couple of ideas for other processing options you might consider implementing in MultiBuffer's ProcessingProc:

- Summation: Averaging the samples of two waveforms will produce a third sample that sounds like both being played simultaneously. This technique can be useful for spitting two input sources out of one sound channel.
- Reverb and delay: These commonly used studio effects are produced by summing the sample points at t and $t+\partial t$,where ∂t is the desired intensity of the effect. The reason I've lumped these two together is that reverb can be considered a really short delay —generally less than 50 milliseconds.

A FEW WORDS ON DEBUGGING

Conditional compilation flags. If you take a look at BuildMultiBuffer, the makefile for MultiBuffer, you'll see -Debug and -Verbose as possible compilation options. These are triggers for Debug messages that can be compiled into the code to help you figure out where the code is failing during development. The advantage of this scheme is that when undefined, the Assert and DebugMessage macros evaluate to (void), which the compiler happily optimizes out.

The general form of this kind of macro is

```
#if _TRIGGER
   #define SomeFunction(s)      SomeFunctionOfS (s)
#else
   #define SomeFunction(s)      ((void)  0)
#endif
```

These can be really useful if you want to generate different types of executables without resorting to multiple parallel source files.

Why debugging interrupt routines is painful. There are two significant problems with debugging interrupt routines. Using a debugger that leaves interrupts enabled (such as TMON) means that you may not be able to observe a "steady state" of your code. Using a debugger that disables interrupts (such as MacsBug) allows you to see a snapshot of the system, but you run the risk of killing the system if you stay in the debugger for any length of time.

As you can imagine, this can make getting an accurate picture of what's really happening nearly impossible. As a result, the approach used in MultiBuffer was to leave a "bread crumb" in the debugger so that one could go back later and see what happened. You'll notice that in routines that are going to be executed at interrupt time the debugging statements have the form

```
Assert (err, "\p Some Message; g");
DebugMessage ("\p Another Message; g");
```

This causes MacsBug to log a copy of "Some Message" in its window and return immediately to the routine. This gets you out of MacsBug quickly enough to avoid causing problems. From the log left in MacsBug, you can often narrow down the cause of the problem. TMON, to the best of my knowledge, doesn't have a similar feature, so MacsBug is really the tool of choice for debugging MultiBuffer. Be careful when running debug versions of MultiBuffer with TMON installed, as Assert and DebugMessage statements executed at interrupt level can leave you in the debugger.

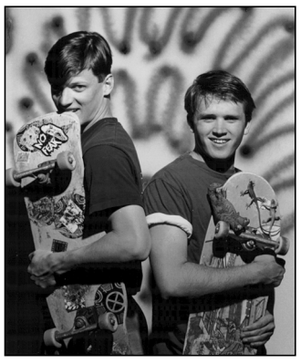
The message you should be especially watchful for is "Tried to queue a buffer that wasn't ready." This indicates that QueueFrame attempted to play a buffer that wasn't marked kBufferReady, and usually indicates that you're taking too much time in your ReadProc or ProcessingProc. The solution here is to either increase the size of your buffers or reduce the amount of time you spend in your read and processing routines.

ALL PLAYED OUT ("KBUFFEREMPTY")

MultiBuffer provides one example of how to play sounds that are too large to fit in memory. Its theory of operation is very similar to that of the Sound Manager routine SndPlayDoubleBuffer in that the buffer-refreshing mechanism keys off the play completion routine; this class of buffering algorithm is appropriate for play-from-disk and related applications.

Adding sound support to your application can greatly enhance your user's experience. Once you have an understanding of a few simple principles, it's also fairly simple. To that end, I hope that MultiBuffer is useful in enhancing your understanding of some of these issues.

THANKS TO OUR TECHNICAL REVIEWERS
Rich Collyer, Leo Degen, Jim Mensch, Jim Reekes•



GRAPHICAL TRUFFLES

WRITING DIRECTLY TO THE SCREEN

BRIGHAM STEVENS AND BILL GUSCHWAN

Many developers want to go beyond the speed of QuickDraw. Writing directly to the screen can allow you to create faster animation and graphics than possible with QuickDraw. However, Apple has always maintained that writing to video memory is unsupported, since it may cause your application to break on a future system. If you write directly to the screen, your application will forfeit the use of many Toolbox managers and will put future compatibility at risk. Since most applications require the Window Manager and other basic Macintosh managers, writing to the screen is only for a few specialized applications, such as video games and some animation packages that compete on the quality and speed of graphics.

We're providing guidelines for writing to the screen in this column because we know that some developers are already doing it. We also understand that, in today's market, you need every advantage you can get in order to be competitive.

BEFORE YOU READ ON

The most important thing to remember is *don't write directly to the screen if you don't have to.* In general, only a few applications need to do this. If you're porting an existing graphics or animation library from another system, or writing an application that competes mainly on the speed of the graphics, writing directly to the screen may be necessary. For any other applications, turn back now and forget about writing to the screen.

Even if your application is animation intensive or a port from another system, we recommend that you always attempt to use QuickDraw first. QuickDraw may be fast enough for your purposes, and it would not be wise to sacrifice its compatibility and flexibility for no reason. You should always have a QuickDraw version of your code anyway, and it should be the default, in case your program isn't compatible with the system or video card being used. Writing directly to the screen should be a user-selectable option.

As an alternative to writing to the screen, your application may be able to increase graphics performance by using custom drawing routines in a GWorld and CopyBits to transfer your image to the screen. This allows you to have faster graphics while avoiding the compatibility nightmare that you may face by writing directly to the screen. To learn more about custom drawing routines, see "Drawing in GWorlds for Speed and Versatility" in *develop* Issue 10.

We hope we've scared almost everyone away. For those of you still reading, we want to point out that violating one compatibility guideline doesn't mean your program should break others: you still need to follow certain rules in order to peacefully coexist with other applications. For example, don't assume the screen is a fixed size or depth. Use data structures like GDevice and screenBits to access this information (*Inside Macintosh* Volume VI, page 3-7).

So remember, most applications have no need to write directly to the screen, and if you choose to do it, it may give you more compatibility headaches than you're ready for. If your program breaks in the future because you decided to write to the screen, it will be *your* responsibility to fix it. We feel that the methods outlined in this column will give you the best chance of future compatibility; however, there are no guarantees.

WHERE'D THAT MANAGER GO?

In addition to risking compatibility problems, writing directly to the screen means you have to do a lot of extra work. Specifically, you have to handle (or live

BRIGHAM STEVENS (AppleLink: BRIGHAM) escaped from mainframe hell to work for Apple in June 1991 on the HyperCard® IIGS project. (He's the one on the right in the photo.) After a short and amazingly entertaining stint writing XCMDs, he joined Developer Support as a contractor in November 1991, and has never been home since. You can tell when you're getting near Brigham's office, because you'll be ducking Nerf arrows, and the sounds of "Dude!" will be raging across your lobes. At night you can find him there basking in the cathode rays of his 16" color monitor. By day you can find him romancing the sidewalk with his skateboard. On weekends he may be dancing in San Francisco, enmeshed in the rhythm of something relentless and metallic. He says one of his weirdest dreams was missing a turn while driving, and then setting up a 68000 jump table to return. His next goal is to be in a PowerBook commercial, on his skateboard saying "Dude, it's the next thing!" while doing an axle grind over a DOS PC—all this with a PowerBook in his right hand running his favorite application, MacsBug.•

without) many of the tasks that Toolbox managers would normally handle for you.

You lose the full benefit of QuickDraw's graphics routines, most importantly the clipping ability. Because the Window Manager uses QuickDraw for its clipping, you lose the ability to have multiple overlapping windows as part of your application's interface. If your application requires multiple overlapping windows, you don't want to be writing directly to the screen.

You lose the ability to stretch your windows across multiple monitors. QuickDraw automagically has the ability to split the contents of a window across multiple monitors. If you write directly to the screen, you'll be limited to one monitor, or you'll have to write a lot of code that has already been implemented in QuickDraw.

You lose the Help Manager. The Help Manager displays its balloons in a window over your application's window. If you're writing directly to the screen, you'll blast the Help Manager's windows.

You lose QuickDraw's ability to map pictures and pixMaps from one color environment to another. Replacing such code with your own is nontrivial. Just try writing an image-copying routine that deals with simultaneous multiple pixel depths and you'll gain a new respect for CopyBits!

You restrict your ability to print. The Printing Manager only understands QuickDraw. To print you'll have to use your drawing code to render your images and then use CopyBits to transfer them to the printing grafPort. This means sacrificing quality on the printed page, since pixMaps generally don't look as nice on the printer as objects composed of QuickDraw calls. Of course, if you have a QuickDraw version of your code you can easily work around this.

Your program may have a different look and feel than a standard Macintosh application. In the case of video games and other animation packages, this may be OK. But if you're writing the next-generation word processor or spreadsheet application, you should be using QuickDraw, the Window Manager, and the Palette Manager. You lose all or most of these user interface managers if you write directly to the screen. Do your writing to the screen within a window; this will lessen the user interface impact. If you want to take over the entire screen, open a window that covers the entire screen.

Your program will also need to know when it's running in the background (see *Inside Macintosh* Volume VI, page 5-19). When you're in the background, other applications' windows may be covering your window. In this case, you *must* use QuickDraw to refresh your window when you get an update event. If you write directly to the screen, you may clobber the foreground application's window.

FEELIN' THE NEED FOR SPEED

Writing directly to the screen for faster animation or graphics means a lot of work for you, because it will be up to your programming skills to beat QuickDraw. QuickDraw does everything possible to be as fast as it can while still being very generic. Efficient use of QuickDraw may actually eliminate the need to write to the screen.

If you write specialized code that's tailored specifically for the kinds of images and graphics that your application deals with, you might be able to make it faster than QuickDraw. The routine presented at the end of this column, which simply draws a color icon, is about 50% faster than CopyBits. It was timed on a Macintosh LC II at 1221 frames per second. CopyBits came in at a relatively sluggish 579 frames per second. The timing was done by taking the average number of frames per second for 100 seconds.

Our routine is faster because it doesn't make any of the extra checks that QuickDraw must make to work with different bit depths and color environments. This routine is also coded to copy an image of a specific size, while CopyBits is a generic bit copier.

Writing directly to the screen doesn't guarantee that you'll be faster than QuickDraw. Even if your code is

BILL GUSCHWAN, reflecting his lack of belief in a self, quotes Wittgenstein: "Whereof one cannot speak thereof one must be silent." Bill enjoys the wrath of Peleus's son, Smashing Pumpkins, Skinny Puppy, and Smashing Candy, which are his favorite book theme, rock groups, and poem, respectively. On a Shakespearean note, he quotes the tenet, "For O, for O, the hobbyhorse is forgot!" or, deconstructed, "Language deceives; never trust it." We think Bill's brain needs a little deconstruction; we could donate it to science because we now know how unselfish he really is.•

To learn how to influence the speed of CopyBits, see the Macintosh Technical Note "Of Time and Space and _CopyBits" (formerly #277).•

good, under some circumstances QuickDraw may outperform you. If there's an accelerator in the system, you may not be able to beat QuickDraw at all. Also, CopyBits is more efficient when copying large images, because the overhead is a smaller part of the overall work.

Increasing drawing speed enhances certain special effects. Not only can you gain smoother animation, but you also may be able to perform complex transformations, such as rotating 3-D shapes, and photo-realistic shading, which QuickDraw can't do. And you can do it with a high animation frame rate. For applications such as games, increased drawing speed and improved special effects are essential.

The table below shows the trade-off between image size and drawing speed, comparing writing directly to the screen, QuickDraw CopyBits, and CopyBits in QuickDraw accelerated by an 8•24 GC card. We copied an 8-bit color image from an offscreen GWorld to the screen, on a Macintosh IIfx with System 7.0.1.

	512 x 384 Image	256 x 192 Image	128 x 96 Image
Writing to the Screen	20	77	306
CopyBits	19	72	268
8•24 GC CopyBits	92	304	770

Numbers shown are frames per second.

THE TEN COMMANDMENTS OF WRITING TO THE SCREEN

1. Be sure your code is faster than QuickDraw.
2. Have a QuickDraw version of your code for compatibility.
3. Write your code for a specific QuickDraw version.
4. Write your code specifically for the kind of data you're dealing with.
5. Write your code for a specific pixel depth.
6. Never change the pixel depth without the user's permission.
7. Always bracket your drawing code with ShieldCursor and ShowCursor.
8. Always draw into a window.
9. Never draw directly to the screen while you're in the background. Use QuickDraw instead.
10. Don't write off the edge of video memory.

GETTING READY

Before you venture into video memory, you should do a few things to prepare for writing directly to the screen. Your program should determine the pixel depth and open a window to draw into. Additionally, for maximum performance under System 7, you may want to kill all applications in the background.

To beat QuickDraw, your code should be tailored for a specific pixel depth. Your program should find the pixel depth of the screen it's writing to by accessing the screen's GDevice.gdPMap.pixelSize field. If the depth is different from the depth that your program expects, you should ask the user's permission to change the pixel depth, and then change it using SetDepth (*Inside Macintosh* Volume VI, page 21-23).

You *must* open a window to cover the part of the screen you're drawing on. If you don't use a window, update events for applications in the background may interfere with your graphics.

If you require maximum performance, and you don't want any applications taking away your cycles at WaitNextEvent time, you may want to consider using the System 7 Process Manager to kill all the background applications. If you do this, you should ask

The code used to measure the trade-off between image size and drawing speed can be found on the *Developer CD Series* disc.•

the user's permission first. Your users will no longer be able to access any desk accessories, which require the Finder to launch. There's an example of killing the background applications, called KillEveryOneButMe, on the *Developer CD Series disc* in the Snippets folder.

REACHING FIRST BASE

To access video memory, you need the base address of the video buffer for the screen to which you're writing. Depending on the version of QuickDraw installed, you'll need to use a different method of getting the base address. Use the Gestalt function with the gestaltQuickdrawVersion selector to determine the QuickDraw version.

- If you're running on a system with the original black-and-white QuickDraw, which has a Gestalt result of gestaltOriginalQDxx, you can access the address of the screen by using the QuickDraw global variable screenBits.baseAddr.
- If you're running Color QuickDraw, which has a Gestalt result of gestalt8BitQD, you can get the base address of the screen you're drawing on from the baseAddr field of the GDevice's pixMap. You should leave the addressing mode alone.
- If you're running 32-Bit QuickDraw, which has a Gestalt result of gestalt32BitQDxx, you can get the 32-bit–clean base address of the screen's video memory by calling GetPixBaseAddr on the screen's GDevice's pixMap. Before you begin writing to the screen, you'll need to shift to 32-bit addressing mode to access video memory (see SwapMMUMode, *Inside Macintosh* Volume V, page 593). After you change the addressing mode, you can't make any Toolbox or OS calls until you switch back, because they all expect to be called with the addressing mode established when the Macintosh was booted. Also, if you're changing modes from 24 bit to 32 bit, you should call StripAddress on any master pointers that may be dereferenced in 32-bit mode, because the high byte may contain garbage.

On multiple-monitor systems, you have to decide whether you want your windows to be on any of the additional screens. If you decide to allow users to drag your window to other monitors, you should get the base address from the monitor that the window is on. See Graphical Truffles in *develop* Issue 10 for a discussion of multiple monitors, including code.

PIXEL ACCESS

Now that you have the base address of video memory, you need to know how to get to a pixel. To access a pixel within video memory, you need to translate the screen coordinate into a byte address.

To map the vertical coordinate, multiply it by the rowBytes and add this product to the base address of the screen's video memory; this gives you the row address. To map the horizontal coordinate, calculate the byte number and add it to the row address. Voilà! You now have the pixel address.

Below is a formula to translate a pixel coordinate into a pixel address. We leave it up to you to implement these formulas in the most efficient way for the graphics you're working with.

```
rowAddr = screenBaseAddr + (rowBytes
            * pixel_vertical_coordinate);
if (pixel_depth < 8) {
   pixels_per_byte = 8/pixel_depth;
   byteNum = pixel_horizontal_coordinate
            / pixels_per_byte;
}
else {
   bytes_per_pixel = pixel_depth/8;
   byteNum = pixel_horizontal_coordinate
            * bytes_per_pixel;
}
pixelAddr = rowAddr + byteNum;
```

For the example at the end of this column, we use the formula for a pixel depth of 8 because it's the simplest. In this case, the above calculations can be reduced to

```
rowAddr = screenBaseAddr + (rowBytes *
               pixel_vertical_coordinate);
pixelAddr = rowAddr +
               pixel_horizontal_coordinate;
```


For information about killing the Finder, see the Q & A on page 115.•

For more about 32-bit addressing, see *develop* Issue 6, page 36.•

MOUSETRAP

You must call ShieldCursor (*Inside Macintosh* Volume I, page 474) before you actually start writing to video memory. If you don't call ShieldCursor, your application will not be compatible with some third-party monitors. Also, if you don't have the cursor hidden and the mouse is moving over an area as you draw to it, the mouse will leave behind "serially repeating artifacts" (garbage). When you're done writing to video memory, call ShowCursor (*Inside Macintosh* Volume I, page 168) to reverse the effects of ShieldCursor.

CONCLUSION

Writing directly to the screen is risky because the Macintosh hardware or OS may change in the future; Apple makes no guarantee that these methods will always work. In writing directly to the screen, you'll sacrifice the services of many Toolbox managers. For mainstream applications, QuickDraw's speed and flexibility will suffice. But for certain applications, such as games and animation, writing directly to the screen may provide an extra competitive edge.

AN EXAMPLE

Below is a sample function, DirectPlotColorIcon, that draws an 8-bit color icon (an 'icl8' resource) to an 8-bit color screen device whose pixMap is passed.

REFERENCES

- "Drawing in GWorlds for Speed and Versatility" by Konstantin Othmer and Mike Reed, *develop* Issue 10.
- "Making the Most of Color on 1-Bit Devices" by Konstantin Othmer and Daniel Lipton, *develop* Issue 9.
- "QuickDraw's CopyBits Procedure: Better Than Ever in System 7.0" by Konstantin Othmer, *develop* Issue 6.
- "Realistic Color for Real-World Applications" by Bruce Leak, *develop* Issue 1.
- "Compatibility: Rules of the Road" by Dave Radcliffe, *develop* Issue 1.
- Macintosh Technical Note "Of Time and Space and _CopyBits" (formerly #277).
- Macintosh Technical Note "Compatibility: Why & How" (formerly #117).
- "Accessing Hardware," *Inside Macintosh* Volume VI, page 3-7.


```
void DirectPlotColorIcon(long *colorIconPtr, PixMapHandle screenPixMap, short row, short col)
{
   register long *screenMemPtr;     // Pointer to video memory
   register short numRowsToCopy;    // Rows we're going to copy
   register short stripRowBytes;    // To clear high bit of rowBytes
   register short rowLongsOffset;   // rowBytes converted to long
   char  mmuMode;                   // 32-bit mode required
   Rect  cursRect;                  // Rectangle for ShieldCursor call
   Point cursOffset;                // 0,0 to indicate rect is in global coordinates

   /* High bit of pixMap rowBytes must be cleared. */
   stripRowBytes = (0x7FFF & (**screenPixMap).rowBytes);

   /* Strip high byte of icon address to prevent bus error in 24-bit mode. */
   colorIconPtr = (long *)StripAddress(colorIconPtr);
```

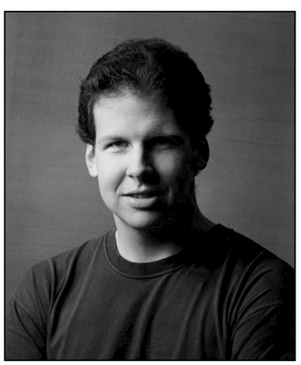

What about that 0x7FFF rowBytes? The largest rowBytes of a pixMap passed into CopyBits is 0x3FFE, otherwise the largest rowBytes is 0x7FFE.•

```
    /* Calculate the address of the first byte of the destination. */
    screenMemPtr = (long *)(GetPixBaseAddr(screenPixMap) + (stripRowBytes * row) + col);

    /* Call ShieldCursor to maintain compatibility with all displays. */
    cursRect.top = row;
    cursRect.left = col;
    cursRect.bottom = row + 32;
    cursRect.right = col + 32;
    cursOffset.h = 0;
    cursOffset.v = 0;
    ShieldCursor(&cursRect, cursOffset);

    /* Change to 32-bit addressing mode to access video memory. The previous addressing mode
       is returned in mmuMode for restoring later. */
    mmuMode = true32b;
    SwapMMUMode(&mmuMode);

    /* Color icons have 32 rows. */
    numRowsToCopy = 32;

    /* Calculate the long word offset from the end of one row of the color icon on the screen's
       pixMap to the first byte of the icon in the next row. */
    rowLongsOffset = (stripRowBytes/4) - 8;

    /* Draw the color icon directly to the screen. */
    do {
          *screenMemPtr++ = *colorIconPtr++;
          *screenMemPtr++ = *colorIconPtr++;
          *screenMemPtr++ = *colorIconPtr++;
          *screenMemPtr++ = *colorIconPtr++;
          *screenMemPtr++ = *colorIconPtr++;
          *screenMemPtr++ = *colorIconPtr++;
          *screenMemPtr++ = *colorIconPtr++;
          *screenMemPtr++ = *colorIconPtr++;

       /* Bump to start of next row. */
          screenMemPtr += rowLongsOffset;
    } while(--numRowsToCopy);

    /* Restore addressing mode back to what it was. */
    SwapMMUMode(&mmuMode);

    ShowCursor();
}
```


Thanks to C. K. Haun, Dennis Hescox, Guillermo Ortiz, and Forrest Tanaka for reviewing this column.•

LIVING IN AN EXCEPTIONAL WORLD

Handling exceptions is a difficult but important part of developing Macintosh applications. This article provides a methodology as well as a set of C tools for handling exceptions and writing robust code. Techniques and examples are provided for dealing with some of the Toolbox idiosyncrasies, and some interesting features of the C preprocessor, MacsBug, and MPW are explored.



SEAN PARENT

Writing software on the Macintosh can be difficult. Writing robust software on the Macintosh is even more difficult. Every call to the Toolbox is a potential source of a bug and there are too many cases to handle—what if there isn't enough memory, or the disk containing the code has been ejected, or there isn't enough stack space, or the printer is unplugged, or . . . The list goes on, and a well-written application is expected to handle every case—always recovering without loss of information. By looking at how software is developed, this article introduces a methodology and tools for handling the exceptional cases with minimal impact on the code that handles the task at hand.

VERSION 1: NORMAL FLOW OF CONTROL

When writing code, programmers usually begin by writing the normal flow of control—no error handling. The code shown below is a reconstruction of the first version of a printing loop routine that eventually went out as a Macintosh Technical Note, "A Printing Loop That Cares . . ." (#161). Note that comments were removed to make the structure more apparent.

```
#include <Printing.h>
#include <Resources.h>
#include <Memory.h>

void PrintStuff(void)
{
   GrafPtr      oldPort;
```


SEAN PARENT (AppleLink PARENT, Internet parent@apple.com) *is* a parent, but Parent is his last name, not his title. He grew up in Renton, Washington, with his parents (you know, the people who produced him), who are also Parents. Sean came to Apple to pursue his lifelong interest in reference manuals. He enjoys a good ANSI standards document during breakfast, and likes catchy punch lines such as, "No, no! I said 'ANSI,' not 'ASCII'!" Sean also likes to write a good hack, and consistently comes in next-to-second-best at the annual MacHack MacHax Hack Contest. Unable to hide his prowess, he gave in to the inevitable job at Apple, and now he wants to change the world, one programming paradigm at a time.•

```
short       copies, firstPage, lastPage, numCopies, printmgrsResFile,
            realNumberOfPagesInDoc, pageNumber;
DialogPtr   printingStatusDialog;
THPrint     thePrRecHdl;
TPPrPort    thePrPort;
TPrStatus   theStatus;

GetPort(&oldPort);
UnLoadTheWorld();
thePrRecHdl = (THPrint)NewHandle(sizeof(TPrint));
PrOpen();
printmgrsResFile = CurResFile();
PrintDefault(thePrRecHdl);
if (PrStlDialog(thePrRecHdl)) {
   realNumberOfPagesInDoc = DetermineNumberOfPagesInDoc(
      (**thePrRecHdl).prInfo.rPage);
   if (PrJobDialog(thePrRecHdl)) {
      numCopies = (**thePrRecHdl).prJob.iCopies;
      firstPage = (**thePrRecHdl).prJob.iFstPage;
      lastPage = (**thePrRecHdl).prJob.iLstPage;
      (**thePrRecHdl).prJob.iFstPage = 1;
      (**thePrRecHdl).prJob.iLstPage = 9999;
      if (realNumberOfPagesInDoc < lastPage) {
         lastPage = realNumberOfPagesInDoc;
      }
      printingStatusDialog =
         GetNewDialog(257, nil, (WindowPtr) -1);
      for (copies = 1; copies <= numCopies; copies++) {
         (**thePrRecHdl).prJob.pIdleProc = CheckMyPrintDialogButton;
         UseResFile(printmgrsResFile);
         thePrPort = PrOpenDoc(thePrRecHdl, nil, nil);
         pageNumber = firstPage;
         while (pageNumber <= lastPage) {
            PrOpenPage(thePrPort, nil);
            DrawStuff((**thePrRecHdl).prInfo.rPage,
               (GrafPtr)thePrPort, pageNumber);
            PrClosePage(thePrPort);
            ++pageNumber;
         }
         PrCloseDoc(thePrPort);
      }
      if ((**thePrRecHdl).prJob.bJDocLoop == bSpoolLoop) {
         PrPicFile(thePrRecHdl, nil, nil, nil, &theStatus);
      }
   }
}
```

```
   PrClose();
   DisposeHandle((Handle)thePrRecHdl);
   DisposeDialog(printingStatusDialog);
   SetPort(oldPort);
} /* PrintStuff */
```

VERSION 2: ERROR HANDLING ADDED

With code in the preliminary stage shown above, the flow of control is easy to follow. After writing it, the programmer probably read through it and added some error-handling code. Adding "if (error == noErr)" logic wasn't difficult, but it took some thought to determine how to handle the cleanup and deal with the two loops. Some more error-handling code may have been added after running the routine under stressful conditions. Perhaps it was reviewed by lots of people before it went out as a Technical Note. Here's the new version of the code (with the added error-handling code shown in bold):

```
#include <Printing.h>
#include <Resources.h>
#include <Memory.h>

void PrintStuff(void)
{
   GrafPtr      oldPort;
   short        copies, firstPage, lastPage, numCopies, printmgrsResFile,
                realNumberOfPagesInDoc, pageNumber, printError;
   DialogPtr    printingStatusDialog;
   THPrint      thePrRecHdl;
   TPPrPort     thePrPort;
   TPrStatus    theStatus;

   GetPort(&oldPort);
   UnLoadTheWorld();
   thePrRecHdl = (THPrint)NewHandle(sizeof(TPrint));
   if (MemError() == noErr && thePrRecHdl != nil) {
      PrOpen();
      if (PrError() == noErr) {
         printmgrsResFile = CurResFile();
         PrintDefault(thePrRecHdl);
         if (PrError() == noErr) {
            if (PrStlDialog(thePrRecHdl)) {
               realNumberOfPagesInDoc = DetermineNumberOfPagesInDoc(
                  (**thePrRecHdl).prInfo.rPage);
               if (PrJobDialog(thePrRecHdl)) {
                  numCopies = (**thePrRecHdl).prJob.iCopies;
```

```
                        firstPage = (**thePrRecHdl).prJob.iFstPage;
                        lastPage = (**thePrRecHdl).prJob.iLstPage;
                        (**thePrRecHdl).prJob.iFstPage = 1;
                        (**thePrRecHdl).prJob.iLstPage = 9999;
                        if (realNumberOfPagesInDoc < lastPage) {
                           lastPage = realNumberOfPagesInDoc;
                        }
                        printingStatusDialog =
                           GetNewDialog(257, nil, (WindowPtr) -1);
                        for (copies = 1; copies <= numCopies; copies++) {
                           (**thePrRecHdl).prJob.pIdleProc =
                              CheckMyPrintDialogButton;
                           UseResFile(printmgrsResFile);
                           thePrPort = PrOpenDoc(thePrRecHdl, nil, nil);
                           if (PrError() == noErr) {
                              pageNumber = firstPage;
                              while (pageNumber <= lastPage &&
                                    PrError() == noErr) {
                                 PrOpenPage(thePrPort, nil);
                                 if (PrError() == noErr) {
                                    DrawStuff((**thePrRecHdl).prInfo.rPage,
                                       (GrafPtr)thePrPort, pageNumber);
                                 }
                                 PrClosePage(thePrPort);
                                 ++pageNumber;
                              }
                           }
                           PrCloseDoc(thePrPort);
                        }
                     } else PrSetError(iPrAbort);
                  } else PrSetError(iPrAbort);
               }
            }
            if (((**thePrRecHdl).prJob.bJDocLoop == bSpoolLoop) &&
                  (PrError() == noErr)) {
               PrPicFile(thePrRecHdl, nil, nil, nil, &theStatus);
            }
            printError = PrError();
            PrClose();
            if (printError != noErr) PostPrintingErrors(printError);
         }
         if (thePrRecHdl != nil) DisposeHandle((Handle)thePrRecHdl);
         if (printingStatusDialog != nil) DisposeDialog(printingStatusDialog);
         SetPort(oldPort);
      } /* PrintStuff */
```

Can you easily follow the normal flow of control in the second version? What if an error occurs? Could an error ever go unreported? Could this code crash because it didn't handle an error? Does this routine always clean up after itself? These questions are difficult to answer because the normal flow of control of this routine is intertwined with the flow that will occur in the event of an error. If the two could be separated, it would be much easier to tell what the routine does normally and how things are handled when something goes wrong. Besides making the code easier to read, a methodology that allowed such separation would make the code easier to write, debug, and, maintain.

PROGRAMMING BY CONTRACT

Programming by contract is based on the assumption that all correct routines have a contract, either stated or implied, with their caller. The contract states that if a given set of preconditions is met, the routine either succeeds or flags an exception and leaves the machine in a known or determinable state. This contract is flexible enough to be applied to any correct code.

The secret to writing robust code is to understand what the preconditions of a given routine are, when an exception can be flagged, and how to handle the exception. Separating the logic that checks conditions and handles exceptions from the algorithm of the routine allows code to be written in a straightforward way with the flow of control seen as easily as in our first version.

PRECONDITIONS

The preconditions of a routine specify the state the machine must be in for the routine to execute without failure (where failure implies a crash—not flagging an exception). A routine may require in its precondition items such as

- a previously called initialization routine
- valid ranges for value parameters
- available memory
- initialization of global state
- a specific software version

For some routines the preconditions may be readily apparent either in the interface or in the documentation. Sometimes it's necessary to experiment to discover the preconditions of a routine. When writing a routine, the "strength" of the precondition can be set according to the use of the routine. For example, a routine named DivideLong is written with a description that states:

> Given two numbers, numer and denom, DivideLong will divide numer by denom, set numer to the result, and return noErr. If denom is zero, DivideLong will return divideByZeroErr and leave numer unchanged.

With this description, numer and denom can be any numbers of the proper type. This is a weak precondition. Another description might read:

> Given two numbers, numer and denom, DivideLong will divide numer by denom and return the result. If denom is zero, DivideLong will fail.

With the second description, it would be the caller's responsibility to ensure that denom isn't zero. This is a strong precondition.

In general, it's better to have a strong precondition in a routine that is used within a sequence of related routines or shares conditions with other routines, because it generates more efficient code by eliminating error checking. It's better to have a weak precondition in routines that are called only once or at the start of a sequence of related routines. Routines with weak preconditions free the caller from ensuring the state of the machine before making the call.

A precondition can be strengthened by the caller but must not be weakened. Strengthening is useful when you're making a sequence of related calls where being sure additional conditions are met guarantees that no routine flags an exception. For example, given the first description of DivideLong it would be valid for a caller to do the following:

```
if (denom != 0) {
   (void)DivideLong(&numer_a, denom);     /* Ignore return. */
   (void)DivideLong(&numer_b, denom);     /* Ignore return. */
} else HandleError();
```

This may be more desirable than checking for a result of divideByZeroErr after each call. An example of weakening a precondition would be to call DivideLong as described in the second description without ensuring that denom isn't zero. This would constitute a bug.

POST-CONDITIONS

Post-conditions specify the state of the machine on the return of a routine. They include side effects and changes to global state as well as function results and variable parameters. The post-conditions of a routine must be determinable for the routine to be correct. They don't vary in strength and, if not met, the routine has a bug. A thorough understanding of the post-conditions of a routine is required to ensure that the routine is being called correctly and that cleanup can occur when the routine flags an exception.

Sometimes it's necessary to rephrase the preconditions and post-conditions of a routine to use it correctly. For example, a common misconception is that the only preconditions for calling TEKey are that it has passed a valid TEHandle and the appropriate Managers have been initialized. Since there's no mechanism for TEKey

to flag an exception, the assumption is that it can't fail. But TEKey may need to grow the hText handle if the character isn't replacing others and isn't a backspace. Growing a handle requires memory—something there may not be enough of. Since TEKey can fail without flagging an exception with these preconditions, it appears to be incorrect and contain a bug. However, by strengthening the preconditions to require that hText must be able to grow by the size of a character, the routine is once again correct. Strengthening preconditions is an easy fix often used in system software. (See the section "Preflighting Calls" for tips on how to ensure preconditions.)

HOW TO WRITE CHECKS

The **check** macro is used to ensure that static preconditions and post-conditions are being met during development. It also documents conditions for you, making it a very useful tool that adds to the maintainability of the code. Unfortunately, these conditions cannot be expressed directly in the interface so as to be more apparent to the caller. The syntax for **check** is

check(*assertion***);**

To use the **check** macro, include Exceptions.h (provided on the *Developer CD Series* disc). For MacsBug, use ResEdit to add Exceptions.rsrc to the DebuggerPrefs file in the System Folder.

What **check** does depends on the setting of the compile-time variable DEBUGLEVEL. DEBUGLEVEL can be set to one of the following values:

- DEBUGOFF or DEBUGWARN: **check** does nothing and *assertion* is not evaluated.
- DEBUGMIN or DEBUGSYM: *assertion* is evaluated and, if it's false (zero), a debugger break is executed. (The debugger break is Debugger() for DEBUGMIN and SysBreak() for DEBUGSYM. The first is useful for low-level debuggers like MacsBug or TMON, the second for symbolic debuggers like SourceBug, SADE, or THINK C.)
- DEBUGON or DEBUGFULL: *assertion* is evaluated and, if it's false (zero), MacsBug is entered and the dprintf dcmd is invoked to display more information. If DEBUGON, *assertion* is displayed and if DEBUGFULL, the source code file and line number are also displayed (see "Wonders of MacsBug and dprintf" for more information about dprintf).

Normally, **check** is used at the start and end of a routine. At the start it's used to ensure that parameters are within a given range and are not specific values (such as nil). At the end it's used to ensure that allocations succeeded and results are as desired.

WONDERS OF MACSBUG AND DPRINTF

The MacsBug dcmd, dprintf, is used by the **require** and **check** macros to display useful debugging information. The dprintf command is also a powerful tool that provides all the features of the standard printf but uses MacsBug as the console. The dprintf command assumes MPW parameter-passing conventions. The syntax for dprintf is

dprintf([**no**]**trace,** *formatString***,** ...**);**

where **trace** and **notrace** are used to specify whether or not to continue after displaying the information in MacsBug. The variable *formatString* is a printf style-format string with some extensions (see the comment in the Exceptions.h file on the *Developer CD Series* disc). Following *formatString* are the parameters to display. This can be a very useful tool for viewing complex structures or difficult-to-read values like floating- or fixed-point numbers.

The implementation of the dprintf dcmd is shown in the DPrintf.c file on the CD. It's fairly straightforward and can be extended easily to add any special data types required (for example, a **t** format character that would take a pointer to text and an integer length and display the text). The dcmd is invoked from C using the inline declaration for dprintf. The inline declaration invokes the DebugStr trap and pushes a long on the stack. The push is required because DebugStr uses Pascal calling conventions and so pops the [**no**]**trace** string from the stack. Since dprintf is a C-based function, the stack is fixed, so the string isn't popped twice. Both **trace** and **notrace** are macro Pascal strings containing ";dprintf;doTrace" and ";dprintf". Since the strings begin with a semicolon, MacsBug interprets them as commands and executes them. The dcmd then fetches the parameters from the stack according to *formatString* and displays them. The MacsBug macro **doTrace** evaluates to "g" or "". It's used to switch tracing between **trace** and **break** by entering either **traceGo** or **traceBreak** in MacsBug.

When developing software, it's useful to insert dprintf statements to display information in sections of code that are executed only in unusual circumstances. If dprintf is bracketed with #if debugon / #endif directives, it compiles out when DEBUGLEVEL is set to DEBUGWARN or DEBUGOFF. With **trace** the information is displayed without seriously interrupting the execution of the code. The **trace** macro is also useful for logging timing statistics by displaying Ticks. Since *formatString* is interpreted in MacsBug with interrupts disabled, even a complex *formatString* has minimal impact on timing results.

MACSBUG POWER USER TIP

If you have more than one monitor, you can use the swap command to make MacsBug always visible and use dprintf with **trace** to continually log information. You can set which screen MacsBug uses by opening the Monitors control panel, holding down the Option key, and dragging the "Happy Macintosh" to the monitor on which you want to display MacsBug (you have to restart for it to take effect).

MPW POWER USER TIP

At the end of the comment for dprintf in Exceptions.h is a section that uses Echo to pipe code to the assembler.

```
/*******************************************
Echo "                                       ∂n∂
   PRINT     OFF,NOHDR                       ∂n∂
   INCLUDE   'Traps.a'                       ∂n∂
   PRINT     ON                              ∂n∂
   PROC                                      ∂n∂
   _DebugStr                                 ∂n∂
   SUBQ       #4,SP      ; Fix the stack     ∂n∂
   ENDPROC                                   ∂n∂
   END                                       ∂n∂
" | Asm -l
*******************************************/
```

If you select this section and press Enter, it generates a listing with hex output. This is a handy way to generate and document inline functions.

REQUIREMENTS FOR BETTER LIVING

Although **check** can ensure that preconditions and post-conditions are being met during development, **check** is of limited value in situations where it cannot be determined whether the conditions are being met statically, because

- it disappears when DEBUGLEVEL is set to DEBUGOFF
- it doesn't provide sufficient support for handling exceptions to return the machine to a known state

What's needed is a mechanism that does not compile out and provides the ability to invoke a handler when *assertion* fails.

WHAT WE REQUIRE

The **require** macro was created to make handling exceptions simpler. The syntax for **require** is

require(*assertion***,** *exception***);**

If *assertion* evaluates to false (zero), execution continues at the handler *exception*. (The *exception* parameter, by convention, shares the name of the routine that failed, but this isn't mandatory.) Handlers are typically written as shells with control falling from one to the next, cleaning up after prior calls along the way. The extent of the cleanup needed gets deeper as more of the routine succeeds. Figure 1 shows an extended form of **require** called **require_action**. The extended form executes a statement when *assertion* fails before executing the handler. This is most useful for setting an error variable. The syntax for **require_action** is

require_action(*assertion***,** *exception***,** *action***);**

Like **check**, **require** breaks into MacsBug and displays pertinent information depending on the settings of DEBUGLEVEL. Unlike **check**, **require** does not compile out when DEBUGLEVEL is set to DEBUGWARN or DEBUGOFF. It evaluates *assertion* and invokes the handler (and *action*), but no break occurs.

The **nrequire** macro is equivalent to **require(!***assertion***,** *exception***)**. However, under rare circumstances it generates more efficient code, and when debugging is on, it displays the value of *assertion*. It's also easier to read. As a general rule, use **require** with handles and pointers and **nrequire** with errors.

VERSION 3: IMPROVED WITH REQUIRE

A close look at the code in version 2 reveals some problems:

- No error handling is done after PrCloseDoc, though any errors will get caught either after the next PrOpenDoc or on exit.

```
OSErr CreateContent(WindowPtr window)
{
    OSErr       error;
    long        scrapSize, tempLong;
    HContent    content;

    check_action(window, return(nilParamErr););

    content = (HContent)NewHandle(sizeof(SContent));
    require_action(content, NewHandle_content,
        error = MemErr(););

    (*content)->picture = (PictHandle)NewHandle(0);
    require_action((*content)->picture, NewHandle_picture,
        error = MemErr (););

    scrapSize = GetScrap((Handle)(*content)->picture,
        'PICT', &tempLong);
    require_action((scrapSize >= noErr) || (scrapSize ==
        noTypeErr), GetScrap, error = (OSErr)scrapSize;);

    if (scrapSize == noTypeErr) {
        DisposeHandle((Handle)(*content)->picture);
        (*content)->picture = nil;
    }

    SetWRefCon(window, (long)content);

    return(noErr);

GetScrap:
    DisposeHandle((Handle)(*content)->picture);
NewHandle_picture:
    DisposeHandle((Handle)content);
NewHandle_content:
    return(error);
} /* CreateContent */
```

Figure 1
Control Flow for **require_action**

- No error handling is done for GetNewDialog; if it fails it may result in a crash.
- If the NewHandle at the start of the code fails, it won't print and the user is never notified why.
- If an error occurs in the copies loop, the loop isn't terminated.

If printing is well behaved and does nothing once PrError has been set, none of these problems poses much of a threat to the actual stability of the code (with the exception of GetNewDialog). However, the use of **require** when writing the code could have avoided the problems and the code would be easier to understand and maintain. This is shown in the code below—version 3. The structure of the code in version 3 is almost identical to version 1 with the addition of the **require** statements and the handlers at the end. Writing code like this is straightforward. When a routine is called that can flag an exception, a **require** statement is added with a handler. The statements executed in a handler typically clean up after the routines called before the routine flagging the exception. (See the section "When To Clean Up" for more discussion.) Although PrClose should never cause an error, a **check** statement was added during development.

Here's version 3 of the code (with changes from version 1 shown in bold):

```
#include <Printing.h>
#include <Resources.h>
#include <Memory.h>
#include <Errors.h>
#include "Exceptions.h"

void PrintStuff(void)
{
   GrafPtr      oldPort;
   short        copies, firstPage, lastPage, numCopies, printmgrsResFile,
                realNumberOfPagesInDoc, pageNumber;
   DialogPtr    printingStatusDialog;
   OSErr        theError;
   THPrint      thePrRecHdl;
   TPPrPort     thePrPort;
   TPrStatus    theStatus;
   long         contig, total;

   enum { dialogSlop = 8192 };

   GetPort(&oldPort);
   UnLoadTheWorld();

   thePrRecHdl = (THPrint)NewHandle(sizeof(TPrint));
   require_action(thePrRecHdl, NewHandle, theError = MemError(););

   PrOpen();
   nrequire(theError = PrError(), PrOpen);

   printmgrsResFile = CurResFile();
```

```
PrintDefault(thePrRecHdl);
nrequire(theError = PrError(), PrintDefault);

if (PrStlDialog(thePrRecHdl)) {
   realNumberOfPagesInDoc = DetermineNumberOfPagesInDoc(
      (**thePrRecHdl).prInfo.rPage);
   if (PrJobDialog(thePrRecHdl)) {
      numCopies = (**thePrRecHdl).prJob.iCopies;
      firstPage = (**thePrRecHdl).prJob.iFstPage;
      lastPage = (**thePrRecHdl).prJob.iLstPage;
      (**thePrRecHdl).prJob.iFstPage = 1;
      (**thePrRecHdl).prJob.iLstPage = 9999;
      if (realNumberOfPagesInDoc < lastPage) {
         lastPage = realNumberOfPagesInDoc;
      }

      PurgeSpace(&total, &contig);
      require_action(contig >= dialogSlop, PurgeSpace,
         theError = memFullErr;);

      printingStatusDialog =
         GetNewDialog(257, nil, (WindowPtr) -1);
      require_action(printingStatusDialog, GetNewDialog,
         theError = memFullErr;);

      for (copies = 1; copies <= numCopies; copies++) {
         (**thePrRecHdl).prJob.pIdleProc = CheckMyPrintDialogButton;
         UseResFile(printmgrsResFile);
         thePrPort = PrOpenDoc(thePrRecHdl, nil, nil);
         nrequire(theError = PrError(), PrOpenDoc);

         pageNumber = firstPage;
         while (pageNumber <= lastPage) {
            PrOpenPage(thePrPort, nil);
            nrequire(theError = PrError(), PrOpenPage);

            DrawStuff((**thePrRecHdl).prInfo.rPage,
               (GrafPtr)thePrPort, pageNumber);
            PrClosePage(thePrPort);
            nrequire(theError = PrError(), PrClosePage);

            ++pageNumber;
         }
         PrCloseDoc(thePrPort);
         nrequire(theError = PrError(), PrCloseDoc);
      }
```

```
            if ((**thePrRecHdl).prJob.bJDocLoop == bSpoolLoop) {
               PrPicFile(thePrRecHdl, nil, nil, nil, &theStatus);
               nrequire(theError = PrError(), PrPicFile);
            }
         }
      }

      PrClose();
      ncheck(PrError());

      DisposeHandle((Handle)thePrRecHdl);
      DisposeDialog(printingStatusDialog);
      SetPort(oldPort);
      return;

PrOpenPage:
      PrClosePage(thePrPort);
PrClosePage:
PrOpenDoc:
      PrCloseDoc(thePrPort);
PrPicFile:
PrCloseDoc:
      DisposeDialog(printingStatusDialog);
GetNewDialog:
PurgeSpace:
PrintDefault:
PrOpen:
      PrClose();
      DisposeHandle((Handle)thePrRecHdl);
NewHandle:
      SetPort(oldPort);
      PostPrintingErrors(theError);
} /* PrintStuff */
```

PREFLIGHTING CALLS

Preflighting a call is the process of ensuring that the preconditions are met. Usually this isn't necessary since the preconditions will be satisfied by handling the exceptions of prior calls or will be implicit in the caller's preconditions. For example, there's no need to ensure that the TEHandle being passed to TEKey isn't nil if the exceptional case of the previous TENew returning nil was handled.

In case preconditions haven't been satisfied by handling the exceptions of previous calls, **require** can be used to check the precondition and invoke a handler if it's not being met. This is especially useful for routines that have strong preconditions or

preconditions that are difficult to determine. Earlier, TEKey was used as an example of a routine with strong preconditions. To ensure the preconditions for TEKey, **require** could be used as follows:

```
OSErr SafeTEKey(short key, TEHandle hTE) {
   enum { teSlop = 1024 };

   OSErr     error;
   TEPtr     w           = *hTE;
   Handle    hText       = w->hText;
   short     teLength    = w->teLength;

   SetHandleSize(hText, teLength + teSlop);
   nrequire(error = MemError(), SetHandleSize);
   SetHandleSize(hText, teLength);
   TEKey(key, hTE);
   return(noErr);

SetHandleSize:
   return error;
}
```

The constant teSlop is used instead of 1 just to be safe. Adding some slop for routines with implied, rather than stated, preconditions is always a good idea.

For some routines the preconditions are too complex or subject to change to accurately state as an assertion. This is the case for GetNewDialog, as shown in version 3. GetNewDialog can fail when there isn't enough memory for one of the numerous QuickDraw elements to be allocated, to load the WDEF, or, if the dialog contains TextEdit items, to create the TEHandle. About all that can be done to guarantee that GetNewDialog succeeds is to ensure that there's a reasonable amount of memory available. It's fairly safe to rely on the Process Manager in System 7 to make sure there's space in the system heap for the WDEF. This is what's done in version 3. The assertion is based on contiguous memory instead of total memory in case the heap is too fragmented to allocate some of the larger blocks required. Sometimes all that can be done is to increase the chances of survival.

WHEN TO CLEAN UP

Just as preconditions can sometimes be tricky to determine, post-conditions can be hazardous as well. It's important to understand the post-conditions of the routines being called, so that the machine can be returned to a known state, ensuring valid post-conditions for the calling routine. Normally, if an exception is being raised, a routine should dispose of everything it *successfully* allocated, close everything it *successfully* opened, and release everything it *successfully* locked. So, if NewHandle is

CouldDialog, which was intended as a preflight tool for GetNewDialog, is a no-op in System 7 (it's been broken since the Macintosh II was introduced).•

called successfully, DisposeHandle is called in the handler. If OpenFile is called successfully, CloseFile is called in the handler.

But this rule isn't always true. One counterexample is the Printing Manager. Even if PrOpen flags an exception PrClose *must* be called. The same is true for PrOpenDocument and PrOpenPage.

Shared resources present another potential problem. If GetResource is successfully called on a system resource, it's a bad idea to release it, because it may also be in use by another routine. SetResLoad(false) and GetResource can be called to determine whether the resource is already in memory before loading it, and then it can be released only if it was loaded. This, however, is taking things to an extreme. It may be better to document that these resources may be loaded even if the routine flags an exception. Since this is determinable by the caller, it suffices as a valid post-condition.

FUTURE DIRECTIONS

The routines and macros provided in Exceptions.h lay the foundation for writing robust software. There are more sophisticated exception-handling mechanisms, such as the proposed **catch** and **throw** implementation for C++. Ada has a reasonable exception-handling mechanism, as does CLU and Eiffel. However, these mechanisms don't lend themselves to dealing with exceptions from routines that were not written using the same mechanism and so are difficult to use on the Macintosh when dealing with the OS and Toolbox. The **check** and **require** macros are flexible enough to be useful in most situations and are implemented in C, so they can be of value for many (if not most) existing projects. They are also C++ friendly and can be of great use to C++ programmers as well.

After you read the code in version 3 that uses these macros it should be fairly simple to answer the questions asked about version 2 at the beginning of the article. This is left as an exercise.

Turn the page if you want even more detail . . .

RELATED READING

- Macintosh Technical Note "A Printing Loop That Cares . . ." (formerly #161).
- *Object-Oriented Software Construction* by Bertrand Meyer (Prentice-Hall, 1988). Contains more information on programming by contract.
- *Debugging Macintosh Software with MacsBug* by Konstantin Othmer and Jim Straus (Addison-Wesley, 1991). Contains additional MacsBug tips.

MORE DETAIL THAN MOST FOLKS NEED

The **require** and **check** macro implementation is shown in Figure 2. To ensure that there aren't any side effects, any macro that's larger than a single statement is enclosed in **do { } while(false)**. This ensures that the macro behaves as a simple statement and can be used anyplace a simple statement would be (such as after an **if**). The **do { } while(false)** does not generate any object code. In some of the macros, **if** statements appear in the form

```
if (assertion) ; /* Do nothing. */
else { /* Do something. */ }
```

Under some conditions, this will generate more efficient code than

```
if (!assertion) { /* Do something. */ }
```

(This was especially true back in the days of the MPW 3.1 compiler.) There are variables declared within the scope of the macros when debugging is on. This avoids side effects caused by evaluating *assertion* multiple times (once in the condition and once to display it). For example:

```
nrequire(ReadCharacters(), Fail);
```

If ReadCharacters returned a value other than nil, MacsBug would be invoked to display the result before executing the handler Fail. Without the local variable, ReadCharacters would be executed a second time to display the value. The second execution may cause side effects like increasing a file pointer as well as reading in a different set of characters.

When *assertion* is an error code returned by a function, it can be assigned to a variable to preserve the error. This also keeps the exception-handling code enclosed within the **require** statement. For example:

```
nrequire(error = GetError(), Fail);
```

However, with warnings set to full, this invokes a warning because the assignment takes place as part of an **if** statement. Using

```
error = GetError();
nrequire(error, Fail);
```

generates identical code (at least with MPW 3.2) and doesn't cause any warnings.

A macro, **resume**, is provided for recovering from exceptions. It's used within a handler and takes the form

resume(*exception***);**

where *exception* corresponds to *exception* used in a **require** statement. The **resume** macro simply transfers control to the point immediately following the **require** statement. Because of the **resume** feature, multiple **require** statements cannot share the same exception handler. Sometimes sharing a handler is convenient, so **resume** can be disabled with a statement:

```
#define resumeLabel(exception)
```

To reenable **resume**, use

```
#define resumeLabel(exception)\
   resume_ ## exception:
```

There's also a **check_action** macro which, like **require_action**, allows a statement to be executed when *assertion* fails. The **check_action** macro compiles out like all **check** macros and should be viewed as a development-time tool only. Being able to execute a statement allows for the exit of a routine if the preconditions aren't met.

Seriously insane cycle counters take note that the MPW 3.2 C compiler doesn't reuse a register to store a variable in a local scope when the register was used in a prior scope containing a **goto** statement (**require** generates a **goto**). This can lead to code that isn't as efficient as it should be but can usually be coded around (it's difficult to generate in the first place). Hopefully this will be fixed in a future compiler.•

```
#define require(assertion, exception)          \
   do {                                        \
      if (assertion) ;                         \
      else {                                   \
         dprintf(notrace,                      \
            "Assertion \"%s\" Failed\n"        \
            "Exception \"%s\" Raised",         \
            #assertion, #exception);           \
         goto exception;                       \
         resumeLabel(exception);               \
      }                                        \
   } while (false)

#define check(assertion)                       \
   do {                                        \
      if (assertion) ;                         \
      else {                                   \
         dprintf(notrace,                      \
            "Assertion \"%s\" Failed",         \
            #assertion);                       \
      }                                        \
   } while (false)
```

Figure 2
Implementing **require** and **check**

THANKS TO OUR TECHNICAL REVIEWERS
Scott Boyd, Konstantin Othmer, Sam Weiss. Also, special thanks to everyone in the Print Shop (present and former members) for using this stuff and suggesting numerous improvements during the past few years.•

THE NETWORK PROJECT: DISTRIBUTED COMPUTING ON THE MACINTOSH

Distributed computing is the wave of the future, soon to come rolling onto the shores of programming. Programmers should be prepared for the possibilities and challenges that distributed computing will offer. The NetWork model proposes a design strategy and provides a testbed implementation that enables you to explore and experiment with distributed computing on the Macintosh. While this article may not help you write a better application today, it will help familiarize you with the idea of distributed computing so that when system support for it comes along, you'll be ready to take advantage of it.

GÜNTHER SAWITZKI

As computing evolves, we're rapidly moving from a reliance on discrete personal computers and workstations to a new type of computing infrastructure—a *computing environment*. In a computing environment, applications will make massive use of many partially coordinated or uncoordinated autonomous computing devices. That is, one device won't necessarily know which application subtask any other device is working on or when and how any other device is completing its particular subtask. These autonomous devices will be connected by multiple threads of communication. What's more, the computing environment of tomorrow will be continually changing, with portable devices moving in and out and with new capabilities added dynamically. Devices will change in time and will have varying availability. In short, distributed computing in an environment with no guaranteed stability will become the order of the day.

Visions like Apple's Personal Digital Assistant and the TRON Project give some idea of what we'll see. The Personal Digital Assistant will be a small intelligent device that will help you with some aspect of living and working; for example, it might be a smart map leading you around in a town you're visiting, or a dietary assistant helping you plan a week's meals, or a TV viewer helping you trace back a thread of interesting news you've just become aware of. TRON will work the other way, making your environment smart on its own; for example, the washing machine itself will place

GÜNTHER SAWITZKI sold his car seven years ago and hasn't regretted it for a second since then. He thinks that cars, along with sports (except for art forms like aikido), are relics of the past. He works (within walking distance of home) at the University of Heidelberg's Institute for Applied Mathematics, doing computational statistics and data analysis when he's not busy with software engineering and development. He headed the NetWork Project and designed the basis of NetWork. In his opinion, Aldous Huxley's *Brave New World* is a vital book of immediate importance. His favorite game is go ("It's the only game that allows me to comprehend that it's a game"), his favorite food is mousse au chocolat (with white and black chocolate), and his favorite time of day is tomorrow.•

orders for more detergent and will tell the warm water supply to diminish for a moment because there will be hot wastewater that will feed a heat exchanger. Both these visions will soon become reality in a distributed computing environment.

What distributed computing will mean for users is that they'll have access to the considerable computing power that's typically left unused in today's computing setup. Implementing a system for distributed computing is easy if you reduce or restrict the availability of personal workstations to their users. The challenge addressed by the NetWork Project is to make access to idle workstations possible while still guaranteeing users immediate access to their personal workstations. NetWork is a minimal communication and management model designed to operate in this environment. By handling communication and managing computing resources, it frees the programmer to think about how to split up a task so that it can be done by multiple workstations working on small pieces in an uncoordinated and asynchronous way.

NetWork is available on the current *Developer CD Series* disc and via Internet for those who want to try it out. This article describes the NetWork Project itself, considers the types of applications that are most amenable to a distributed computing approach, thoroughly examines the NetWork model, and then suggests how to implement a NetWork program on the Macintosh. Because I'm a statistician I've included some discussion of statistical underpinnings. I've presented this discussion separately, though, so that if you don't find mathematics fascinating, you can skip it.

HISTORY OF THE NETWORK PROJECT

NetWork is a project of StatLab, the statistical laboratory at the University of Heidelberg. StatLab was founded in 1984 to complement the existing mathematical statistics research group by studying practical applications of advanced statistical methods. We took a look at what was available as the hardware base for our work and chose the Macintosh, but since no Macintosh was on the German market at that time, we bought a Lisa. We've been developing our statistical software on Lisa and Macintosh ever since. This eventually brought us into contact with Larry Taylor, representing Apple's Advanced Technology Group in Europe.

During a November 1988 meeting, we discussed future perspectives in computing with Larry. We tried to identify current gaps and obvious next steps. One thing we could point to was the discrepancy between the amount of computing power we had installed and the return it gave us. At that time, we were running an installation of Macintosh Plus and Macintosh II computers, and the usual turnaround time for a statistical simulation was one night. This was better than the turnaround time for the same job on the IBM mainframe time-sharing system (about a week), but still it was frustrating to have to wait so long while other computer resources lay idle. Just the same, given the Macintosh's character as an absolutely devoted servant of one master, how in the world could we find a way to share its computing power while still guaranteeing reliable and efficient service for the Macintosh owner?

For more on the TRON Project, see *The TRON Project, 1988: Proceedings of the Fifth TRON Project Symposium.*•

In December 1988 we had a visit from Bill Eddy, then head of the statistics department at Carnegie Mellon University. In a lecture he mentioned that the CMU people were annoyed at the discrepancy between installed computing power and the return it gave them and were doing research on executing iterations asynchronously (in an uncontrolled way) to make use of aggregated computing power. Until then, I'd been thinking of the solution only in terms of distributed computing in a *controlled* environment. Bill emphasized that in the computing environment of the future, computing time per se won't be expensive. In fact, in a network consisting of thousands of CPUs, computing power will be *free*—if you can access it. This started me thinking about how we could possibly make a distributed system work under these circumstances—that is, in a large heterogeneous environment.

When we next met with Larry Taylor in February 1989, I claimed that we could build a system for distributed computing based on the Macintosh philosophy of the absolute priority of the user and at the same time able to cope with a large environment. Larry agreed to support the project, and we formed a team consisting initially of Larry, me, Reimer Kühn and Leo van Hemmen of the Heidelberg Neural Network Research Group, and Joachim Lindenberg, then a computer science student at Karlsruhe University.

The project started in May 1989. We called it the NetWork Project, a reference to the fact that in the future the only measure of performance that will matter will be the *net work done per unit of time*, not cumulative computing time or other measures of resource utilization. We gave ourselves six months to decide on the specifications and build a working prototype of a distributed system that would fit a Macintosh environment and be scalable up to some thousands of CPUs. Although Macintosh was the original development target, we did make sure that the system would run in any other decent environment (DEC™, UNIX®, what have you). We finished our first release one week late in November 1989. As they say, the rest is history.

Worth mentioning is the fact that with NetWork's accelerated development schedule, we didn't spend a lot of time on planning and administration. That's the nature of progress sometimes. Fortunately, Apple's Advanced Technology External Research Group had resources available to allocate to the project on the spot. Without this kind of flexible support, the NetWork Project could not have succeeded.

CANDIDATES FOR DISTRIBUTED COMPUTING

Distributed computing will be a great boon to applications where computing power is critical and where the computing task can be split into discrete subtasks. Such applications include the following:

- compiling a new product using a superoptimizing compiler
- solving an optimization problem like placing chips on a board

- generating computer graphics, especially ray tracing
- performing optical character recognition

In these cases, processing may take too long on one particular machine, but if the application can tap into the computing power available by sending out subtasks, the processing can be completed in a much more timely manner.

Many applications that involve working on large data sets can benefit from additional computing power, even in an environment where completion of a subtask is not guaranteed. Such tasks include sorting with some appropriate merge/sort algorithm: the global sort can benefit if a subset has already been sorted by another machine but need not be affected if the result of the presorting is not available. The same applies to searching and practically all major accounting tasks. Any statistical analysis based on exponential families, like normal (Gaussian) distributions, can also benefit from distributed computing: in these analyses you can calculate global sufficient statistics from those of partial data sets, if available. Problems of this type are completely splittable into subtasks and clearly are fine candidates for distributed computing.

But what about problems that have a stronger internal structure than those that are completely splittable? What about iterative and recursive problems, or problems that lead to pipeline processing or networks of data flow? We can't automatically assume that these can take advantage of additional computing power in a distributed environment where the completion of a subtask isn't guaranteed. Still, mathematical theory can help us identify problems of this type that are good candidates for distributed computing.

A SPECIAL CLASS: ASYNCHRONOUS ITERATIONS

As an example of problems with a stronger internal structure than those that are completely splittable, we'll focus on iterative algorithms. The trouble with running an iterative algorithm in a nonguaranteed distributed environment is this: the outcome of iterations in one part of the problem might critically depend on results from iterations in other parts, and the result of a previous iteration may or may not be available for the next round. Even if the original iteration converges to a correct result, we don't know whether the same will hold true if the iterations are done asynchronously.

Suppose, for instance, we have a mapping to be iterated that operates on some high-dimensional vector or matrix. To prepare for a distributed version, we restrict the mapping to a subset by providing the full input but allowing the mapping to operate only on the coordinates selected by the subset. We allocate different subsets to different machines for a number of iterations. These iterations are performed in parallel. The results are collected as they come in and new tasks based on these results are redistributed repeatedly.

In a guaranteed environment, we could wait for all results to come in before assigning the next round of tasks. But in a nonguaranteed environment, we don't know whether a result will come in, and if it does, when. Synchronizing tasks may be impossible. And even when possible, it would be a waste of computing power, because we would spend much of our time waiting for the latest result to come in. Enter asynchronous iterations. Asynchronous iterations don't spend time on waiting. New tasks are assigned as partial results come in. The only question is, will asynchronous iterations give us a correct result?

Mathematical theory can tell us under what conditions asynchronous iterations will yield correct results in a nonguaranteed distributed environment. According to G. M. Baudet in his paper "Asynchronous Iterative Methods for Multiprocessors" in the *Journal of the ACM*, if the original mapping is what mathematicians call a Lipschitz contraction, in general an asynchronous iteration will converge to the same limit as the original mapping. Many numerical methods can be formulated such that they fall into this class. For example, the time-consuming core in many applications—like solvers for differential equations, optimizers, or matrix inversions—can be implemented as algorithms that correspond to Lipschitz contractions.

AN EXAMPLE: NEURAL NETS

As an example of the use of asynchronous iterations in a distributed computing environment, let's look at a neural net applied to picture reconstruction, from work done jointly by Reimer Kühn and me. The specific variant of neural nets we're using is a Hopfield net. Neural nets provide a useful model for cognitive functions; when we reconstruct a picture using a neural net, we're modeling how humans might recognize someone they know in a blurred photograph.

Kühn and I developed an interactive program for associative recall of visual patterns called Spinning Brain. The program, which is included on the *Developer CD Series* disc, first trains a neural net on a series of pictures. Each pixel in a picture is linked to a neuron in our net. Then rudimentary pictures based on the originals are presented to the net. The program then reconstructs the originals from the rudimentary pictures by iterating a certain transformation until a stable state is reached.

In a distributed computing environment, we can take a slice, represented by a subset of the pixels, and ask an idle workstation to perform a number of transformations on it. The restriction to one slice means that only pixels in that slice can be changed, although the full picture is available as initial information. As illustrated in Figure 1, while one slice is being processed on one workstation, we pass other slices as subtasks to other workstations. When we get a result, we merge the processed slice with the rest of the picture; that is, our updating function uses the processed slice to replace the corresponding part of our original picture. This may introduce an error because the processed slice may depend on the state in other slices, which may have changed significantly in the meantime. We repeat the assignment of tasks until we reach a stable state. This example isn't a Lipschitz contraction and thus isn't covered by

Numerical methods that can be formulated as Lipschitz contractions are discussed in Part 2 of *Parallel and Distributed Computation* by D. P. Bertsekas and J. N. Tsitsiklis. •

Hopfield nets are described in more detail in "Spinning Brain: An Interactive Program for the Associative Recall of Visual Patterns" by R. Kühn and G. Sawitzki and in Chapter 5 of *Brains, Machines, and Mathematics* by M. A. Arbib. •

Baudet's convergence result, but under mild regularity conditions, convergence to the original limit still holds.

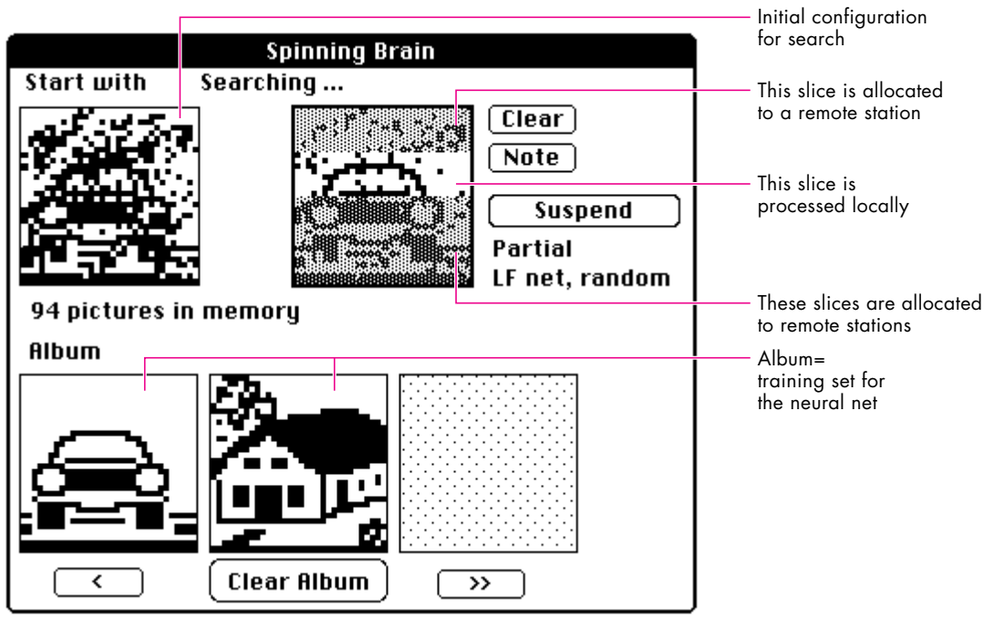


Figure 1
Spinning Brain in Action

Neural nets are an interesting target for asynchronous distributed computing. If we accept that neural nets provide a useful model for cognitive functions, we still must admit that in real biological systems there's no indication of global synchronization except on a very large scale (for example, daily rhythm). Information processing takes place in a distributed asynchronous environment (the brain). And we must admit that this isn't a guaranteed environment: some results may be late or may never be reached. This is true for the individual and even more so for collective or social cognitive phenomena. So experiences with neural nets in our environment might shed light on critical aspects of neural network modeling in an asynchronous, nonguaranteed environment.

THE NETWORK MODEL OF DISTRIBUTED COMPUTING

Now that you know how the NetWork model was developed and have an idea of the kinds of applications that might take advantage of a distributed computing environment, we turn to the model itself. First I'll list the design goals for the

NetWork model; then I'll list the services NetWork needs and the services the Macintosh makes available. From there I'll explain the principles of operation and the layers of the NetWork model. Finally, I'll discuss some important strategies incorporated in the NetWork model to help meet its goals.

DESIGN GOALS

Simply stated, the primary goal of the NetWork model is to make use of the idle resources of a network while respecting the absolute priority of events and processes initiated by each machine's owner. The model implementation runs in an unobtrusive way, making use of free network resources but interfering as little as possible with any user request. The approach we take is to allow other users to borrow the computing power if a machine is idle, but to impose a strict rule: if the owner accesses the machine, the guest is given only minimal time to retreat. The machine has to be completely available without any noticeable delay. This imposes a time to leave of about 1/10th of a second, which might be too short for any proper notification or cleanup.

NetWork takes the view that for every machine there is an owner. The owner may, but need not, correspond to a real user. For example, if the machine is a dedicated server, the server process can be considered the owner. Furthermore, a NetWork machine in general will, but need not, correspond to a physical machine. For example, a cluster of CPUs may be considered a machine for the purposes of NetWork.

Even if there is no immediate owner access, a machine may be busy because an owner-initiated process needs the resources of the machine. The absolute priority of the owner must extend to owner-initiated processes as well. A machine is considered idle, or free for the purposes of NetWork, if there is no owner access and no owner-initiated activity. NetWork is only allowed to use resources that are free in this sense.

The goal to use only free network resources also affects communication. The effect for any owner other than the one requesting network services should be barely noticeable, and care must be taken not to compete for network bandwidth. Unfortunately, with current technology it's nearly impossible to avoid interfering with other users. All that can reasonably be done is to use "second-class" communication where possible and to take measures to minimize the number of network accesses and the additional network load.

To allow for open environments, independence of the underlying communication model (including network/file/bus-based communication, network topology, and such) and adaptability to heterogeneous hardware are additional design goals of NetWork. We aren't narrow-minded: we don't mind making use of a Cray computer via Hyperchannel if it's idle. Finally, to invite experiments with our model, the implementation of an asynchronous iteration scheme should be as near to that of a standard iteration scheme as possible.

In summary, then, the design goals of the NetWork model are as follows:

- immediate availability of any machine to its owner
- minimal interference with owner communication
- independence of communication model
- adaptability to heterogeneous hardware
- close resemblance to a standard iteration scheme

NECESSARY SERVICES

To meet the design goals, NetWork needs the following services:

- idle/busy state monitoring to keep track of owner activity
- process management to launch a process to serve a remote request and to kill all processes launched by NetWork when the owner accesses the machine
- communication to pass message descriptions and results

First, NetWork needs a monitor whose only task is to keep track of whether the machine is idle or whether it's active on behalf of its owner. Since this is machine-specific information, each machine must be equipped with such a monitor, which we call an idle monitor.

Second, NetWork needs a process manager that's capable of handling all process management on remote request. If the machine is idle, the process manager can launch processes to fulfill remote computing requests, and it's responsible for cleaning up all remote processes immediately if the state of the machine changes from idle to busy—that is, if the owner accesses the machine. The process manager is informed of any idle/busy transition by the idle monitor. It's responsible for guarding the priority of the owner. The process manager keeps track of active processes on the local machine.

Third, NetWork needs a communication system. The communication system has to guarantee reliable services in a possibly unreliable environment. Moreover, it should take special precautions to minimize interference with owner communication, as required by the NetWork design goals.

The idle monitor, the process manager, and the communication system form the core of the NetWork model. They must be present in any implementation of NetWork. This core provides convenient primitives for distributed computing while shielding the transport system. In this respect it resembles other approaches, such as those described by G. Bernard, A. Duda, Y. Haddad, and G. Harrus in their article "Primitives for Distributed Computing in a Heterogeneous Local Area Network Environment" and by T. J. Gardner, I. M. Gerard, C. R. Mowers, E. Nemeth, and

R. B. Schnabel in their paper "DPUP: A Distributed Processing Utilities Package." Going beyond these approaches, NetWork tries to provide a minimal model suited even for a nonguaranteed environment.

SERVICES AVAILABLE ON THE MACINTOSH

Given that an idle monitor, a process manager, and a communication system are necessary to the NetWork model, let's look at what we've got on the Macintosh.

The Macintosh doesn't have an idle monitor. If one were available, many applications could take advantage of it. It could relieve applications of the tedious calculations needed to find out which sleep value to use. (Some applications never seem to get this right!) And it would allow a clean strategy for background tasks like indexing and compressing. So we decided that we should implement an idle monitor for NetWork.

Fortunately, the Macintosh Operating System provides an event queue. Since the OS is user oriented, there's a clear model for user events, and all are funneled through the event queue. But looking at the event queue isn't sufficient. A user might have started a time-consuming calculation and left for lunch. In this case, the machine should be considered busy. If it's not, the user might come back and find the machine in slow mode or serving someone else. On the Macintosh, we run a statistic of the CPU program counter to catch these situations. This still leaves frontmost applications that are allowed to consume arbitrary time on the Macintosh. This is where the most important feature of the Macintosh enters: the Human Interface Guidelines. We monitor any cursor changes and busy cursor states to catch this situation as well.

A process manager is available with System 7. This takes care of many tasks that NetWork has to fulfill under previous system software. However, processes under System 7 don't have priority attributes: System 7 can launch processes but doesn't know which processes to kill when the owner comes back. NetWork has to implement this needed functionality. What's more, the System 7 Process Manager is designed to launch an application on a single machine and isn't set up to handle remote launching, so this additional functionality has to be provided by NetWork. To enable portability, NetWork has its own process manager. If you're using System 7, the NetWork process manager maps to the System 7 Process Manager where appropriate and has augmented functionality where necessary.

AppleTalk is the native communication system on the Macintosh. There are restrictions, however. Current implementations of AppleTalk support just one transport system. NetWork has its own communication system, which maps to AppleTalk if appropriate but isn't restricted to AppleTalk. With NetWork's communication system you can talk UDP from the TCP/IP suite to one machine while engaging in AppleTalk with another one. NetWork supports any number of concurrent transport systems, with no gateway needed. And the NetWork communication system tries to reduce additional communication load that would compete with immediate users.

NetWork's communication system is message based. We wanted our message-passing system to be as flexible and powerful as possible. In particular, we wanted it to have extremely low overhead, we didn't want it to impose unnecessary size limitations, and we didn't want it to be restricted to certain operating systems or transport systems. For these reasons, we decided to use our own message-passing system, instead of using Apple events.

For the Macintosh, we've bundled the idle monitor, the process manager, and the communication system kernel into a control panel extension, the NetWork Processor. To use NetWork, you move the NetWork Processor into your System Folder and restart your Macintosh. Programmers can access the NetWork services with the help of a library (NetWorkLib.o) and interface files that come with NetWork. For tips on how to use NetWork's idle monitor and communication system, see "Cheap Thrills: Using NetWork's Services."

NETWORK LAYERS AND PRINCIPLES OF OPERATION

NetWork views the computing environment as a set of machines with processes running on them. Each machine has an owner, who has absolute priority on this machine. Processes can run on behalf of the (local) owner, or they can satisfy a remote request. If a process is running on behalf of a remote request, it should be terminated immediately when the owner accesses the machine. A process handles tasks and eventually may generate tasks for remote execution. A task can be delegated to another process, possibly on a different machine, and results may or may not be returned.

The NetWork programming model has three layers, as shown in Figure 2. The top layer, the application layer, contains the application-specific code. Apart from initialization and cleanup sections, this code should be able to define subtasks and to handle results from subtasks if available. The specific details of this layer are, of course, application dependent.

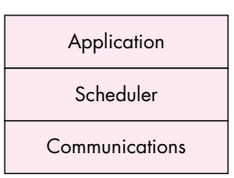


Figure 2
Layers of the NetWork Programming Model

The scheduler layer provides support for asynchronous iterations. The NetWork scheduler monitors and stimulates the generation, assignment, and integration of subtasks. While the proper generation of subtasks is application dependent, the

CHEAP THRILLS: USING NETWORK'S SERVICES

Even if you don't plan to implement a NetWork system, you might find some of NetWork's services very useful indeed. If you install the NetWork Processor, you can make use of any NetWork service. For example, you can ask NetWork whether your station is to be considered idle instead of implementing all the code yourself.

THRILL 1: USING THE IDLE MONITOR TO HELP YOU EXECUTE A BIG JOB

Move the NetWork Processor into your System Folder and reboot your Macintosh. Modify your code to include NetWork.p and link to NetWorkLib.o.

Add the following line to your initialization code:

```
myErr := InitNetWork(NetWorkEvent);
```

Add the following line to the idle branch of your main event loop:

```
IF Idle THEN DoNextRoundOfMyGreatBigJob;
```

DoNextRoundOfMyGreatBigJob is executed whenever NetWork considers your machine to be idle.

A word of warning: If DoNextRoundOfMyGreatBigJob is compute intensive, this will move your machine to the busy state, so "WHILE Idle DO . . . " would not be a good idea.

THRILL 2: USING THE IDLE MONITOR TO LAUNCH AN IDLE TASK

Move the NetWork Processor into your System Folder. Create a folder named NetWork Idle Tools in your System Folder. Move your application into NetWork Idle Tools. Your application will be launched whenever NetWork considers your machine to be idle. Note that because NetWork will kill any application it has launched when the state of the machine changes to busy, this use of the idle monitor makes sense only for turnkey applications such as screen savers. (See the ScreenSaver example provided with NetWork.)

As NetWork has a chance to learn that the application is not a user-initiated process, the machine will stay in the idle state (in contrast to Thrill 1).

THRILL 3: USING THE COMMUNICATION SYSTEM

Move the NetWork Processor into your System Folder. Modify your code to include NetWork.p and link to NetWorkLib.o.

Add the following line to your initialization code:

```
myErr := InitNetWork(NetWorkEvent);
```

Add the line

```
MyHandleMsg(MsgPtr(Event.message));
```

to your main event loop, like so:

```
CASE Event.what OF
   mouseDown:
      DoMouseDown(Event);
   . . .
   NetWorkEvt:
      MyHandleMsg(MsgPtr(Event.message));
```

Your application will now receive messages from NetWork. You'll have to write the MyHandleMsg procedure to evaluate the messages. Message format and support routines are documented in the *NetWork Programmer's Guide.*

NetWork scheduler can monitor the overall system behavior and try for dynamic load balancing. Task assignment is an interaction between scheduler and application.

The communications layer forms the basis of the NetWork design. It provides the basic communication services needed for the network system. In particular, it provides transport shielding to cope with a potentially unreliable environment. If necessary (for example, to implement diagnostic or management tools), the services of the communication system can be accessed directly, avoiding the scheduler.

NetWork is implemented as a message-passing system. A process may send task descriptions as messages, and results are returned as messages. If a process is set up for task generation, the scheduler will ask the application periodically for the definition of a new task. If a new task definition is given, the scheduler will pass this information to the communication system for further transmission. If a process is set up for result handling, the scheduler will inform the application of any result received by the communication system.

In the NetWork model, messages flow as diagrammed in Figure 3. The task-generating application defines a task message and hands it to the scheduler. The scheduler does the necessary housekeeping and passes the message to the NetWork Processor, which communicates it to the receiving NetWork Processor. The receiving NetWork Processor launches the destination application (if necessary). The destination scheduler passes the message to the task handler of its application.

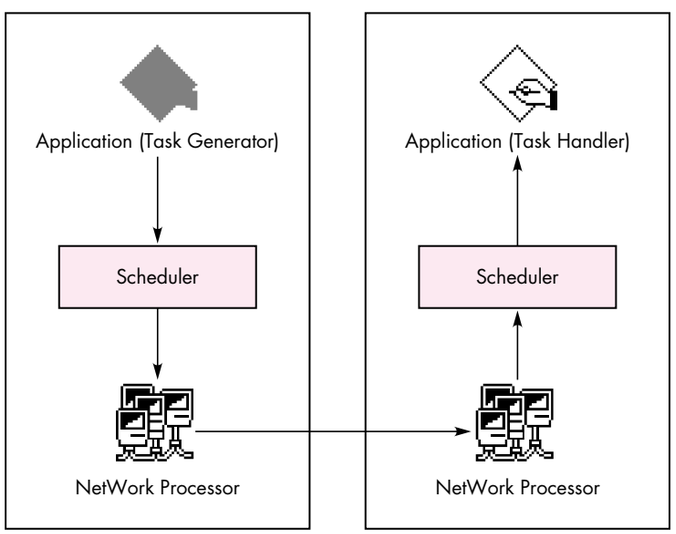


Figure 3
Simplified Diagram of the NetWork Message Flow

Since NetWork is designed to work in a nonguaranteed environment, no assumptions about the lifetime of a communication partner are made. Hence, a process that's generating tasks doesn't know its target in delegating a task. The scheduler proposes a target to which the next task can be delegated when asking for a new task definition. The application is free to accept this proposal or to select a different target using a lookup server or any other source of information.

Messages are addressed to processes, residing on machines. However, in a nonguaranteed environment, no assumption about the existence of a communication partner can be made. The address refers to a process class (defined as any instantiation of the underlying program) rather than to a particular process instance. On the recipient machine, NetWork checks whether the target is active—that is, whether there is a corresponding process. If so, the message is made available. If the machine is idle but no corresponding process is active, NetWork tries to locate the program and launch it first. If it fails, the message is discarded. No prolonged negotiation takes place and no acknowledgment is made. The task message is an implicit launch command, and the completed result is the only acknowledgment, if any. If the state of a machine changes from idle to used—that is, if the owner accesses the machine—NetWork immediately kills any application it has launched.

SOME IMPORTANT STRATEGIES

The NetWork model uses three important strategies to meet its goals effectively. These strategies have to do with minimizing the communication load, recruiting idle machines that are most likely to remain idle, and minimizing the probability of conflicts among incoming messages.

Strategy 1: Minimize the communication load. As stated earlier, one of NetWork's design goals is to minimize the communication load to avoid competing with machine owners. We've already mentioned that NetWork allows a process to be launched implicitly by sending a task addressed to it, and that NetWork avoids negotiations and explicit launch sequences. This is done to reduce additional communication load. Of course, it's possible to use explicit authentication and authorization schemes and exert direct control over launching with NetWork, and in any environment where security is required this will be necessary. But it's in no way required for a minimal implementation of distributed computing, so it's not required in the NetWork model.

The decision not to enforce any session maintenance techniques, nor even any acknowledgment schemes, is another measure to minimize communication load. NetWork can operate in a connectionless mode, so session maintenance techniques or acknowledgment schemes aren't required. Again, if needed, both can be applied.

Since NetWork is designed to work in a noisy environment where no guarantees of availability or performance are given, NetWork has to be prepared for messages that are outdated or out of context. To minimize communication load in these cases,

NetWork encourages a separation of descriptive information from bulk load. Conceptually, each NetWork message consists of a priority part, which should be small and contain just enough information to indicate whether the message is usable in a given context, and the message core, which should contain the bulk of information, as shown in Figure 4. When a message arrives, the priority part along with the usual administrative information is presented to the recipient for inspection. Only if the recipient accepts the message as usable does the core information need to be transported.

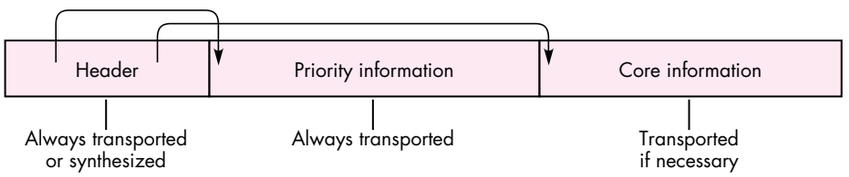


Figure 4
Message Segments

The separation of priority information from core information is only a conceptual one. The NetWork communication manager will do packing/unpacking and transport in a way that seems optimal for the transport system. In particular, for a packet-oriented transport system, the communication manager will pack header and priority information into a first transport system package and fill it up with as much core information as fits reasonably into this package. (Note that the communication manager should signal a received message only if all parts of the priority data are received, but it need not rely on a handshake.) Subsequent packages with the remainder of the core information will be sent only if the recipient requires this information. Thus, unnecessary information load can be avoided. The scheduler included in the NetWork distribution package is adapted to this optimization strategy.

Strategy 2: Recruit the idle machines most likely to remain idle. We need to identify idle machines and have a strategy to allocate them for cooperation. The idle state is determined by the idle monitor, and idle machines can be registered as possible compute servers using a lookup server. Of course, we'd prefer to use those machines that will be available for some time and to avoid those machines that are free for the moment but will be used shortly. To do this, we need some way to distinguish the most promising machines—some method to ascertain what we'll call the *hazard-to-leave-idle-state*.

Our first informal review of literature and interviews with experts gave us little hope of finding some indicator of this hazard. Still, disregarding any recommendations, we implemented an allocation scheme based on observed idle times and then measured

the availability of idle machines. Our results implied that the frequency of useless (short-time) allocation of machines can be drastically reduced by waiting until a certain critical idle time has been exceeded before allocating a task to a particular machine. This is the approach we take in the NetWork implementation. (If you're interested in the details of how we arrived at our conclusion, see "Diagnostic Plots for the Statistically Minded.")

Strategy 3: Filter incoming messages. A scheduler for NetWork can be integrated in applications and make use of the services of the NetWork system. In the current NetWork implementation, a scheduler prototype is provided, together with a library that interfaces with the NetWork communication system. The scheduler asks the application regularly whether a new task should be defined or informs the application of incoming messages. It also does a preliminary check for the usefulness of incoming messages, filtering out messages that can be identified as useless or outdated with respect to the application context.

To guarantee fail-safe behavior, tasks should be allocated redundantly. As a consequence, more than one result may be returned relating to a particular subtask. This poses a problem to the scheduler. Assume we have two incoming partial results. If the first result is based on an earlier state and if less work (fewer iterations) has been done for this result, it's clearly outdated. Or if the first result is based on more recent information and if more work has been invested in this result, it's clearly the better one and should replace the other result. The remaining cases enter a critical region where the scheduler is required to make a decision. (See "Deciding Between Results" if you'd like to read this in mathematical language.)

Our strategy is to accept only those packages that can be accepted without any further analysis. Instead of putting computational power into evaluating the optimal acceptance decision, we try to keep the probability of entering the critical region low. Since our criterion is the time it takes to perform the task, and both acceptance decision making and task allocation are done by the same machine, there's a trade-off between those two, and we can keep the expected loss due to a wrong decision small by keeping the probability of conflicts low.

The NetWork scheduler uses an adaptive task assignment scheme to minimize the probability of these conflicts: from the received results, the scheduler tries to estimate the relative complexity of a subtask and the relative computing power of the partners. New tasks are calibrated so that the expected return time is distributed homogenously, thus reducing the probability of conflicts. An application can override or augment the generic strategy as provided by the scheduler with a more application-specific strategy. In the Spinning Brain example that comes with NetWork, you can see the scheduler trying to adapt to the relative computing power and reliability of the partners. Choose the Scheduler menu item from the Control menu. You'll see a running plot of the task size assigned to machines versus the time of allocation by NetWork, as illustrated in Figure 7.

DIAGNOSTIC PLOTS FOR THE STATISTICALLY MINDED

Read this if you're interested in the details of how we compared the idea the experts gave us about predicting the hazard-to-leave-idle-state versus our own hunch about how it might be predicted.

The general idea we met with was that usable idle time would be controlled by a Poisson process, so the idle time would have an exponential distribution. But since an exponential distribution is memoryless, there would be no chance for optimization based on waiting times: the hazard-to-leave-idle-state would be constant.

To test this idea, we used a special statistical tool—diagnostic plots. Diagnostic plots represent statistics in a way that makes their message easy to grasp. A diagnostic plot is often designed by a statistician in such a way that the significant information shows up as the deviation of a curve from a straight line, visual information that's easy for humans to process.

To find out whether a certain distribution is exponential, we plot observed idle times against those that would be expected given an exponential distribution. If the idle time distribution were in fact near to exponential, this plot would exhibit a straight line. As you can see in Figure 5, this clearly isn't the case.

How we plot the relevant information to test for a Weibull distribution is more complicated, so we won't go into the details here. (Ask your statistician!) Suffice it to say that as shown by the fairly linear behavior of the plot in Figure 6, the idle time distribution is more adequately approximated by a Weibull distribution than by an exponential distribution.

This Weibull distribution has a decreasing hazard rate. For the application this means that it's helpful to know how long a machine has been idle. In particular, the hazard-to-leave-idle-state is lower if a machine has been idle for some time.

So if you have a chance to select among machines, here's the winner's strategy: choose the machine that's been idle for the longest time.

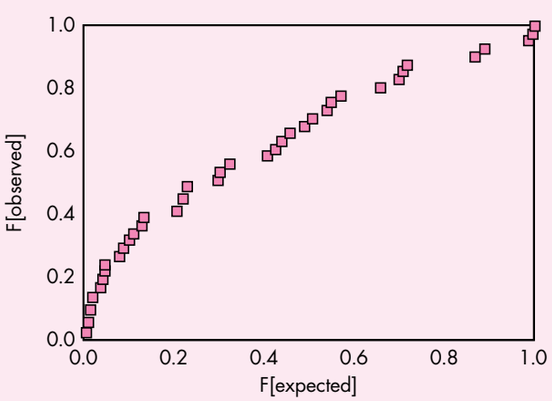


Figure 5
Sample Plot Checking for Exponential Distribution

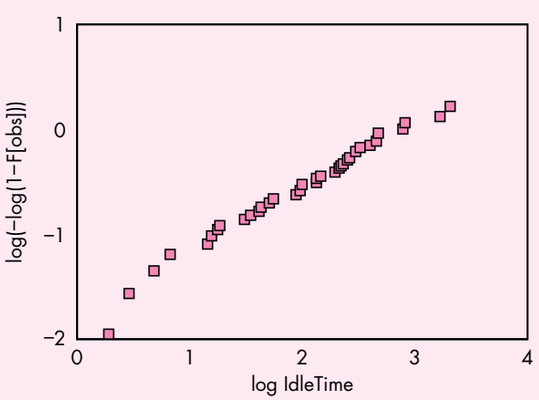


Figure 6
Sample Plot Checking for Weibull Distribution

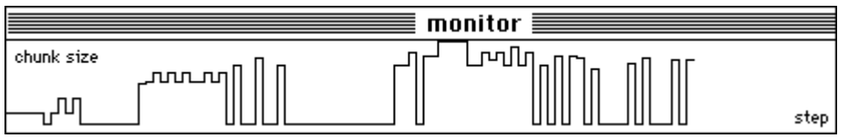


Figure 7
Running Time-Plot of Assigned Task Size, From Spinning Brain

NetWork's ability to adapt itself to the relative computing power of the partners provides a natural way to do load balancing. By finding out the relative performance of the CPUs available and allocating larger tasks to more powerful CPUs, NetWork is able to effectively balance the work load.

HOW TO IMPLEMENT A NETWORK PROGRAM

Now for the good part. You're familiar with the design and operation of NetWork. Here's your chance to explore how your application might make use of distributed computing with the help of NetWork. The following discussion will give you a general idea of how to make your application work with NetWork, but you should study the full example code included with NetWork on the *Developer CD Series* disc for a thorough understanding.

DECIDING BETWEEN RESULTS

For the mathematically minded: Assume we have some effective time scale (some measure of effective iterations done, for example). Assume we have two incoming partial results Y and Y', where Y is based on information available at effective time T, with K iterations done on Y, and Y' is based on information available at time T', with K' iterations. Let Y arrive at time t, Y' at time $t' > t$. Should we replace the results of Y by those of Y'?

There are trivial cases: If $T' < T$ and $K' < K$, then Y' is clearly outdated. Or if $T' > T$ and $K' > K$, then Y' is better than Y, so Y should be replaced. Put another way, results based on better initial information ($K' - K > 0$) and with better iteration count ($T' - T > 0$) can be accepted a priori. Results based on poorer initial information ($K' - K < 0$) and with fewer iteration counts ($T' - T < 0$) can be rejected a priori. For the remaining cases, a decision must be made. Figure 8 shows the limits of the acceptance region. The NetWork strategy is to take only those results that can be accepted a priori.

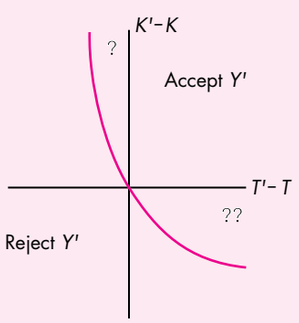


Figure 8
Limits of Acceptance Region for Results

The signature you use when you experiment with NetWork should be NetE (this spelling). This signature has been registered with Apple by the NetWork Project and is reserved for experimental use.•

NetWork will communicate with your code by NetWork events. You have to augment your event-handling code to handle these events. If the what field of the EventRecord is NetWorkEvt, the message field of the EventRecord will contain a pointer to a NetWork message.

```
{******************* The Event Handler *******************}
PROCEDURE DoEvent(Event: EventRecord);
   . . .
   BEGIN
      CASE Event.what OF
         mouseDown:
            DoMouseDown(Event);
         . . .
         {*** You add a case to handle events of type NetWorkEvt. ***}
         NetWorkEvt:
            NetWorkScheduler.HandleMsg(MsgPtr(Event.message));
         . . .
         app4Evt:
         . . .
      END; {case}
   END;
```

To keep NetWork running, you should give it a chance to fulfill its regular tasks, like asking you for new jobs or looking for idle workstations. This should be done in your main event loop. Since we're interested in getting the most from our computing power, we're using a slightly more elaborate event loop than you'll usually find in the DTS Sample Code on the CD. We prefer to calculate the next time to call WaitNextEvent in a more flexible way to get the most from our computing power if our application is frontmost. The next time to call WaitNextEvent will be kept in a global variable gNextEventLoopTime.

```
{******************* The Event Loop ******************}
PROCEDURE MainEventLoop;
   CONST
      cSleep = 0;          {Ticks to wait for wake-up}
      cBackgroundSleep = 20;
      cEventLoopDelay = 1; {3 = 1/20 second, recommended interval between
                            WaitNextEvents for human interaction. We
                            take 1 for faster response.}
   VAR
      newEvent:      EventRecord;   {Event from GetNextEvent}
      hasWNE:        BOOLEAN;
      eventReceived: BOOLEAN;
      mySleep:       LONGINT;
```

```
BEGIN
   hasWNE := system.WNEIsImplemented;
   mySleep := cSleep;   {This is the foreground delay.}
   REPEAT   {Loop until done.}
      IF hasWNE THEN
         BEGIN
         {No mouse moved is wanted, so pass NIL for the mouseRgn.}
         eventReceived := WaitNextEvent(everyEvent, newEvent,
                                        mySleep, NIL);
         UpdateCursor;  {Change the cursor shape if appropriate.}
         END
      ELSE
         BEGIN
         SystemTask;    {Let the system do its stuff.}
         UpdateCursor;  {Change the cursor shape if appropriate.}
         eventReceived := GetNextEvent(everyEvent, newEvent);
         END;
      SetEventLoopTime(cEventLoopDelay);  {Adjust global variable
                                          gNextEventLoopTime.}
      IF eventReceived THEN DoEvent(newEvent)
      ELSE  {No real event, just timeout}
         REPEAT

            {*** You add the following section. ***}
            NetWorkScheduler.PeriodicTask;   {Allow to generate
                                             new tasks.}
            IF NlTask <> noErr THEN    {Try to look up new partners.}
               ProgramBreak('NlTask Error');
            mySleep := NetWorkScheduler.GetSleep; {Adjust sleep value.}
            {*** End of added section ***}

            MyTask(BackContinue, mySleep);   {Do local job.}
         UNTIL (gTaskState <> TaskOK) | (LongIntPtr(Ticks)^ >=
                                          gNextEventLoopTime);
      IF PAbortFlag THEN gTaskState := TaskCancel;
      {PAbortFlag is a function to check whether the standard abort
      combination has been pressed. gTaskState is a global variable
      where we keep the current state of the program.}
      IF gTaskState IN [TaskExit, TaskFatal, TaskAbort] THEN
         gAppDone := TRUE;
   UNTIL gAppDone;
END; {End of main event loop}
```

Of course, accessing global memory locations like Ticks is bad programming; you should use TickCount instead. And you shouldn't do direct comparisons

(LongIntPtr(Ticks) ^ >= gNextEventLoopTime); you should use a function to do comparisons instead. But because this part is in the main loop and we didn't want to waste any time here, we use this dirty inline comparison.

To start NetWork, you have to generate an instance of the scheduler by calling new(NetWorkScheduler) and activate it by calling NetWorkScheduler.init. NetWorkScheduler is defined in the file SchedulerUnit.p that comes with NetWork. If you've activated or used the scheduler, you should always call NetWorkScheduler.free before leaving your program.

If you're going to generate subtasks, you have to override the task generator. Take the prototype definition tTaskGenerator from SchedulerUnit.p and adapt it to your needs. Create a task generator object and call NetWorkScheduler.InitTaskGenerator to install it. To customize a task generator, you have to write a function NewTask. NewTask should return NIL if no subtask can be defined, or a message pointer defining a new subtask. The proper task definition is private to you. The scheduler's task-sending activity can be controlled by NetWorkScheduler.SetSending.

If you think of a master-slave setting, you can implement the code for both sides in one program. At run time, you can use the function Master from the NetWork library to find out whether you're running as master or as slave.

```
{*************** Main Routines ******************}
PROCEDURE MyInit; {(VAR TheState : TaskStateType)}
   VAR
      myTaskHandler:       tTaskHandler;
      myMasterTaskHandler: tMasterTaskHandler; {Used for masters only}
      mySlaveTaskHandler:  tSlaveTaskHandler;  {Used for slaves only}
      myTaskGenerator:     tMyTaskGenerator;   {Typically for masters only}
      myResultHandler:     tReplyResultHandler;
      . . .

   BEGIN
      . . .
      {Initialize the NetWork library.}
      IF InitNetwork(NetWorkEvt) <> noErr THEN fatal;

      {Initialize the name lookup manager.}
      IF NlInit <> noErr THEN fatal;

      {Create and initialize a NetWorkScheduler. Needs a persistent
      memory, so NetWorkScheduler must be a global variable.}
      new(NetWorkScheduler);
      IF NetWorkScheduler = NIL THEN fatal;
      NetWorkScheduler.init;     {The scheduler is up and running now.}
```


Further details on customizing the task generator are given in the *NetWork Programmer's Guide.*•

```
    {Create and initialize a handler for incoming messages.}
    IF NetWorkScheduler.Err = noErr THEN
       BEGIN
       IF Master THEN    {Master is defined in NetWorkLib.}
          BEGIN
          new(myMasterTaskHandler);
          myTaskHandler := tTaskHandler(myMasterTaskHandler);
          END
       ELSE
          BEGIN
          new(mySlaveTaskHandler);
          myTaskHandler := tTaskHandler(mySlaveTaskHandler);
          END;
       IF myTaskHandler <> NIL THEN
          NetWorkScheduler.InitTaskHandler(myTaskHandler);
       END; {End of NetWorkScheduler installation}
    . . .
    {Create and initialize a task generator.}
    IF Master THEN
       BEGIN
       new(myTaskGenerator);
       IF myTaskGenerator <> NIL THEN
          NetWorkScheduler.InitTaskGenerator(myTaskGenerator);
       END;
    . . .
END;
```

Programming for NetWork in general consists of writing a master process (later to be the client seeking additional computing resources) and a compute server. The compute server has to be distributed to the coworkers (the additional computing resources that can be called upon). To guarantee fail-safe behavior, both task generation and task-handling functions should be implemented on the original generating machine so that it can operate by itself if need be. These functions must be implemented in the master process (compute client). Note that to avoid virus proliferation, worms, and other nasty things, NetWork doesn't do any active transportation of code. The code to be launched has to reside on the destination machine and is under the control of the destination owner.

The compute server must be able to accept and handle subtasks. Although it's possible to use the message-handling system of NetWork directly, we recommend you use the supplied scheduler model instead. If you're going to accept subtasks, you have to customize the task handler. Take the prototype definition tTaskHandler and adapt it to your needs. Create a task handler object and install it by calling NetWorkScheduler.InitTaskHandler. To customize a task handler, you have to write a function MsgUsable and a procedure MsgEvaluation. The scheduler will get the

priority information from an incoming message to the PriorityBuffer indicated by MsgPrioPtr. MsgUsable should check any incoming task on the basis of the header information and the available priority information. If MsgUsable returns TRUE, the scheduler asks the message system to pass the bulk of the data describing the subtask to the core buffer indicated by MsgCorePtr. You have to write a procedure MsgEvaluation to take the data from the buffer and initiate the proper task execution. To return a result to the sender, you can make use of the ReplyMessage function.

With NetWork, programs can be launched automatically on remote request. Programs launched on remote request may be terminated by NetWork when the owner accesses the machine. Don't assume it's safe to continue processing at that time if you receive a Command-Q. You must clean up as soon as possible or you won't have another chance. Also note that you don't have the time to report results, because all messages—including those about to be transferred—are killed when your application dies. Remember that NetWork's priority is with the owner, not with your application. The only way to override this is to control the process class of your application. If it's necessary to clean up, set your process type to master after program initialization and call the Idle function regularly. But be forewarned that users may become annoyed at having an alien application around, and your application will likely be removed from the list of welcome visitors.

RISKS IN DISTRIBUTED COMPUTING

Anyone working in distributed computing should be aware of the risks involved in a distributed system. Such risks include those relating to competition for resources as well as those relating to security.

COMPETITION FOR RESOURCES

Any distributed computing system competes for computing and communication resources. NetWork has been designed to minimize the impact of this competition on priority users. Still, the version of NetWork currently distributed uses the AppleTalk Name-Binding Protocol (NBP) to register and look up idle stations, and the AppleTalk NBP is prone to impose a cumulative load that increases with the square of the number of workstations. This will create a problem if the number of workstations in the network is very large. The version of NetWork in distribution won't impose a big load if used in networks with up to 100 workstations. If you do have more workstations in your local zone, please consult the *NetWork Programmer's Guide* for suggestions—our research version scales linearly to accommodate up to 10,000 workstations. If you have more than 10,000 Macintosh computers, we'll have to invest some additional thinking, which we'll gladly do.

Distributed computing systems can also compete for disk space with priority users. This is a crucial point for UNIX-based systems. On a UNIX-based system you can send a guest process to the background, but this still may result in a swapping behavior that's a nuisance to the priority user (unless you're using Mach). For

Further details on customizing the task handler are given in the *NetWork Programmer's Guide.*

NetWork, we decided to kill any guest process if the priority user returns, so NetWork doesn't compete for disk space.

SECURITY CONSIDERATIONS

Other risks relate to the security of code and information. Just as programs and data can carry viruses into a machine from the outside, so distributed computing guests can bring in something undesirable. When you grant access to another user, you never know whether you're enabling the importation of a Trojan horse. For the present, we don't see any way to guarantee system security under conditions of distributed computing, so we've chosen two ad hoc actions to improve security.

First, we refrain from code migration. Of course, it would be most convenient to make use of a remote machine without any assumptions about the availability of code on that machine, and we'd love to do this. But this would require moving executable code if necessary or training the receiving machine on the job. Because we don't see any way to check whether that code contains a virus, the code to be executed is required to be already available to the host machine. Furthermore, NetWork assumes that an access path is denoted on the host machine and launches only applications resting in this trusted path. This path may direct code to a server, and the usual access control mechanisms apply.

Second, we include with NetWork an example called RemoteJob, designed to educate users about the risk of allowing remote execution of powerful code like MPW. Even if there's no virus attached to the code of MPW, it's powerful enough to allow you to compile new programs, viruses and all. The point of including this example is to forewarn you of this possibility. RemoteJob takes commands from the sending station, passes them to the recipient, and launches the MPW shell there if it can be found in the trusted path. The default example passes a "beep" command to MPW, but it could just as well get MPW to compile a virus and install it on the fly. The moral of the story: *Never put a shell or any powerful tool in the trusted access path.*

BACK FROM THE FUTURE

After reading this article you should have a good idea of the possibilities and challenges that are bound to confront programmers with the advent of distributed computing. These possibilities and challenges are already being actively explored in some quarters. In particular, the NetWork model of distributed computing has already been used in a variety of applications. Some examples: a distributed file system using NetWork was built at the University of East Anglia; a U.S. company used NetWork to implement a distributed rendering system; and an IBM subsidiary in France is using NetWork for distributed compilation/program construction.

But for most of the world, the distributed computing wave is still just out there on the horizon. We need to begin playing with and prototyping applications *now* with distributed computing in mind, so that when system support arrives, we'll know how

THANKS TO OUR TECHNICAL REVIEWERS
Michael Gough, Larry Taylor, Peter Zukoski•

to use it. In sum, the time we spend experimenting with NetWork now is sure to pay off in the not-too-distant future when the distributed computing wave comes rolling in.

REFERENCES AND FURTHER READING

- "Asynchronous Iteration" by W. F. Eddy and M. J. Schervish, *Computing Science and Statistics: Proceedings of the 20th Symposium on the Interface, 1987* (American Statistical Association, 1988), pages 165–173.
- "Asynchronous Iterative Methods for Multiprocessors" by G. M. Baudet, *Journal of the ACM* (1978), pages 226–244.
- *Brains, Machines, and Mathematics* by M. A. Arbib (Springer, 1987).
- "DPUP: A Distributed Processing Utilities Package" by T. J. Gardner, I. M. Gerard, C. R. Mowers, E. Nemeth, and R. B. Schnabel, *ACM SIGNUM Newsletters* (1986, Issue 4), pages 5–19.
- "Finding Idle Machines in a Workstation-Based Distributed System" by M. T. Theimer and K. A. Lantz, *IEEE Transactions on Software Engineering* (November 1989), pages 1444–1457.
- *NetWork Communications* by J. Lindenberg (Universität Karlsruhe, Institut für Betriebs und Dialogsysteme, 1990). Republished on the current *Developer CD Series* disc.
- *NetWork Programmer's Guide* by G. Sawitzki (Universität Heidelberg, Institut für Angewandte Mathematik, 1990, 1991). Republished on the current *Developer CD Series* disc.
- *Parallel and Distributed Computation* by D. P. Bertsekas and J. N. Tsitsiklis (Prentice-Hall, 1989).
- "Primitives for Distributed Computing in a Heterogeneous Local Area Network Environment" by G. Bernard, A. Duda, Y. Haddad, and G. Harrus, *IEEE Transactions on Software Engineering* (December 1989), pages 1567–1578.
- "Spinning Brain: An Interactive Program for the Associative Recall of Visual Patterns" by R. Kühn and G. Sawitzki, *Wheels for the Mind (Europe)* (Apple Computer, Inc., January 1989).
- *The TRON Project, 1988: Proceedings of the Fifth TRON Project Symposium* edited by K. Sakamura (Springer, 1989).

FURTHER CREDITS Studying asynchronous iterations in a nonguaranteed (random) environment was suggested by the paper by W. F. Eddy and M. J. Schervish entitled "Asynchronous Iteration." W. Rheinboldt suggested the scheduler strategy of accepting only those packages that can be accepted a priori. The NetWork communication system was designed and implemented by J. Lindenberg.

THE VETERAN NEOPHYTE

QUANTUM LUNCH

DAVE JOHNSON, WITH MICHAEL GREENSPON

I've just read the book Alan Turing: The Enigma, by Andrew Hodges, an outstanding and profound— if thick— biography of Alan Turing. Turing's work touched on some deep philosophical questions about the relationship between brains and computers. I naturally had my own opinions, but I wanted to talk to somebody with more knowledge of brains who was also computer savvy—someone with a foot in both worlds. So I paid a visit to Michael Greenspon, who develops software models of neural systems with Walter Freeman at UC Berkeley. We got together for lunch and had a very interesting conversation. Here's a sample:

[Audio embellishment: clinking of nice glassware as Dave and Michael dine in the sun]

DKJ: I heard something recently that struck me as profound: computers don't manipulate reality, they manipulate *representations* of reality. The profound part is that seems to be what brains do, too. Alan Turing, for much of his life, wanted to build a brain. He firmly believed that consciousness was caused only by the operation of the brain, and that the brain's operation could eventually be described at any level of detail.

[Michael looks patiently skeptical, but Dave plunges ahead, oblivious, waving his fork excitedly.] Further, he had previously proven that in principle, a "universal machine," of which the computer is a finite approximation, could simulate any other logical machine, and thus any logical process whatsoever. So if you could describe the function of the brain as a logical process, you should be able to program a computer to "be" a brain. The description part, of course, is the killer. But I can't help thinking that we'll get there eventually. What do you think?

MCG: Whoa, Dave [almost choking on his exotic Thai salad], I think you've hit the intellectual cul-de-sac of traditional artificial intelligence. The reason it's so hard to describe the operation of the brain as a logical process is simple: it isn't a logical process at all. That's a cerebral approach to a fundamentally biological and physical problem. I'm sure someday we'll be able to logically explain the operation of the brain in terms of physics, but that explanation won't include a computational mechanics based on formal logical operations.

DKJ: But then how do you approach the problem of trying to understand and model brains in your lab, if you can't describe them as logical processes?

MCG: Our approach is that of computational neuroscience; we're doing dynamic modeling at the level of cell populations, using massively parallel machines with a Macintosh front end.

When I say representationalist AI is a cerebral approach, it helps to realize that the cerebral cortex is just a few millimeters thin. It's a tissue essential for generating the separatist intellectual conception of ourselves as humans, but it's really a translucent veneer over the bulk of what our brains do day in, day out, which comes from our animal ancestors. Before we ever learn formal or even natural languages, our brains are already highly developed as processors of spatial, tactile, and kinesthetic information, to name my favorites. This is one reason why the Macintosh has been so successful as a tool—because it's the first readily available machine to offer at least at the outer layer a spatially based interface.

DKJ: And the reason that's so great is that our brains process spatial information effortlessly, without our even trying.

DAVE JOHNSON once borrowed a friend's video camera so that he could spy on his dogs when they were alone. He carefully—and gleefully—set up the camera near the front door, turned it on, and went out for dinner and a movie. The dogs mostly just slept, with an occasional barking fit, apparently just for doggie grins. It was really very dull viewing except for one incident about halfway through: the smallest dog, affectionately known as Dinky, sat down right in front of the camera, stared balefully into the lens for a moment, then put her head back and howled for a full five minutes, something Dave has never seen before or since.•

MCG: Right, a spatial interface allows us to apply more of our innate biological intelligence in communicating with the machine. But both structurally and functionally, the digital computer as a metaphor for the brain is almost completely inaccurate at every level of analysis.

I think if you look further into the nature of thought and perception, and also look more carefully through microscopes and macroscopes at what real brain tissue is doing, you'll see a physical system that exhibits chaotic dynamics in time, has fractal extent in space, and is inextricably linked to the natural world. Computers are powerful tools for simulating and visualizing these properties, but they don't themselves *have* these properties yet.

DKJ: Especially the links to the natural world.

MCG: Exactly. If you want to apply computational metaphors to the brain, perhaps the brain is like a fractal architecture computer that can compute infinitely recursive functions in finite time.

DKJ: Oooh, I like the sound of that. Fractals, computers, infinity, and recursion all at once.

MCG: I like it too, but that's really just a structural metaphor. I'm interested in what we can learn about how real brains might work, so that we can apply these principles to next-generation user interfaces and to new non–von Neumann computing architectures.

In an engineering sense, we're after machine perception. That is, we want future machines to interact in the human sensory world, rather than forcing humans to interact in the virtual world of the machine.

DKJ: Yeah, to use or program a computer today you still have to interact on the machine's terms. I think one good approach to changing that is to try to build computational structures that are like the brain, so that our machines will be a little more like us. There are 10 billion neurons in the brain, more or less, right?

MCG: More. And perhaps 10^{15} synapses, which you could say is where a lot of the computation is going on.

DKJ: OK, more than 10 billion neurons in the brain, and they're wired together in *unbelievably* complex ways. The point is this: I'll bet that we can simulate a single neuron fairly closely with a computer, and over time we can get our simulation closer and closer to the real thing, *arbitrarily* close. Further, I'll bet that someday it will be possible to get 10 billion little computers together and talking to each other. I know this is a little speculative, but my business card says "Limit Pusher," and I feel compelled to live up to it.

MCG: Rave on.

DKJ: So we set this thing up—10 *billion* little processing nodes—and we turn it on and start feeding it information. What will happen? What will it do? I can't help thinking that whatever it is, it will be something very much like life. And just as mysterious.

MCG: Well, I don't think it's purely an issue of scale. At Berkeley, we're building a new ring architecture parallel machine based on superscalar processors that can accommodate multimodal sensors and effectors. It's called CNS-1 and is spec'd at upwards of 100 billion operations per second.

DKJ: 100 BIPS!

MCG: Right. Or 0.1 TRIPS, which is perhaps a better indication of how far we have to go. We expect CNS-1 will be able to simulate many of the emergent dynamical properties of cell populations observed in real brains—to run what I call the lava lamp model of the mind. But even this much power won't bring us "arbitrarily close" to the wetware. I don't think you'll want to say it's alive or that it works the way a biological brain works.

DKJ: Maybe not, but I think that a network of 10 billion processors could *act* something like a brain, could *seem* like a brain, even though it's not one by any stretch of the imagination. That idea fascinates me: that

MICHAEL GREENSPON is a doctoral student in the department of Electrical Engineering and Computer Science at UC Berkeley. When he's not cramming for quals, he can often be overheard trying to explain the cost benefits of telecommuting to Apple managers. (We're still not sure when he sleeps.) If the sun's out, you're sure to find him soaking up some of it; since the release of the Macintosh PowerBook and ToolServer, he's hardly been seen indoors except for an occasional rave. In fact, he and Dave Johnson were recently spotted rigging a LAN in the outfield at Golden Gate Park. He does, however, respond to his e-mail: he can be reached via AppleLink as INTEGRAL or on the Internet as mcg@icsi.berkeley.edu.•

a computer, or a bunch of computers, can behave like something else. This gets back to Turing's thesis that a computer can simulate anything, if you can describe the thing in enough detail. That begs the question, though, of whether the simulation is *fundamentally* the same as the reality it simulates.

MCG: Is it live or is it Memorex?

DKJ: Precisely. It's like comparing painting on the computer to painting using canvas, brushes, and oils. At one level of description they're identical activities: applying color to a surface in intricate and skillful ways to produce a little piece of space that other humans can look at and experience emotion toward. But the tools differ hugely and, perhaps more important, the experience of using them is completely different. So I guess what I'm saying is that at the right level of description I believe (well, I *want* to believe) that it's possible to "build a brain."

MCG: Or to grow a brain. I think you're barking up the wrong dendritic tree. It's *experience* that's essential. Brains are dynamic systems that actively reach out into the sensory world for experience; perception is a creative process, not a passive one. To talk about building a machine with the capabilities of the human brain you have to include the same kinds of connections to the world that humans have. In the real tissue, it goes right down to the level of quantum phenomena and beyond—what I call "real virtuality."

What I've been trying to get across is that real brains operate by virtue of being physically continuous systems; there's an interplay between the nanoscopic and macroscopic, the intrinsic and extrinsic, such that structure and function are not separable. The notion that there exists in brains a "level of description" at which cognition is implemented as logical operations is a convenient fallacy, what John Searle calls "closet dualism." It means, for example, that if you want to start capturing the creative, human aspects of language—not just the literal, but the slang, humorous, ungrammatical, and allusory—you have to model the dynamics of the underlying physical processes.

DKJ: Hmm, this point about not being able to separate cognition from sensory experience is important. It's interesting to compare the development of computers with the development of life. Computer sensors and effectors—the parts of computers that by necessity touch the world—always seem to lag way behind the other parts, the computing parts, in their development. And the gap seems to be widening. So computers are currently wrapped in sensory cellophane, while the connection of biological systems to the world is very strong and high-bandwidth.

MCG: Exactly. It's likely that in biological systems, sensors and effectors developed first and, as part of an evolutionary feedback loop, drove the development of the nervous system. Though now you could say the demands of more sophisticated user interfaces are driving the development of CPUs. The perceptual side is limited to 2-D mouse tracking and 1-D clicks and keystrokes. But speech and pen gestures are about to expand that. Eventually computer-human interface will be polymodal, including intonation, spatial gestures, eye position, facial expression, and cortical activity patterns—what I call the "think-along interface."

DKJ: It fascinates me that programmers can so easily get sucked into the machine—I know *I've* been there—despite the very limited modes of interaction with it.

MCG: Yes, in programming, I often feel I'm being sucked into a one-dimensional world of historical arbitrariness. I think this comes from the fact that while the complexity of our software systems has increased exponentially, our development tools haven't kept pace. The current tools fail to provide the real-time, interactive turnaround that's crucial to maintaining the creative flow. They force us to think too much about the machine's problems, instead of the human problems we're presumably trying to solve.

DKJ: Amen. And it's true for nonprogrammers, too. So how would you like to see the tools improve?

MCG: Well, besides speed—where speed means real-time, no perceptible delay; anything less is slow—

future tools will have semantic knowledge of the process of software engineering and eventually of the application you're building. The code browsers are a good step forward; at least they can automatically determine structure from syntax. The next step is to automate the build process, the incremental linking of components, and the maintenance of an audit trail and nonlinear undo space for source code. Here we start to blur into a dynamic-language sort of model.

DKJ: That's exactly the kind of administrivia that computers are supposed to be good at. But right now, for most of us, the burden is still on the human.

MCG: It sure is. Where we want to head is to shift the focus of the iterative process from the syntax level—compilation, debugging—which is what the machine is concerned with, to the level of design and validation, which is hopefully where the programmer is trying to solve the semantic problems of the application.

DKJ: Way back in the 1940s Turing talked about the fact that ". . . as soon as any technique becomes at all stereotyped it becomes possible to devise a system of instruction tables which will enable the electronic computer to do it for itself." In other words, as soon as we can describe how we do a job, we can program the machine to do it for us. This is happening, but slowly. As an amusing footnote, he went on to say "It may happen however that the masters [programmers] will refuse to do this. They may be unwilling to let their jobs be stolen from them in this way. In that case they would surround the whole of their work with mystery and make excuses, couched in well chosen gibberish . . ." He was a pretty prescient guy.

[Setting his napkin on the table] Well, I guess we should try to wrap it up here; our readers' MacApp builds are probably finished by now, and we'll be losing them soon. Let's try to wring a message out of our ramblings, something developers can take home with them. How about this: Strive to bring computers ever more firmly into the world of people, rather than trying to cram people ever more firmly into the world of computers. The differences can be subtle, but the distinction is very important.

MCG: Well, I think we can and will go much further toward humanizing the experience of using computers. But I don't think we have to couch what we do in gibberish to keep our jobs, because programming is fundamentally a creative discipline. Like other creative disciplines, when you've done it long enough and intently enough, you tend to see its way reflected in everything you perceive. You could say programming is a way of seeing. That leads us to computers as tools for extending human visualization.

[Flipping up his shades] The point is that it's human vision—not the technology—that's crucial. When we create tools and toys and lifestyles that separate and insulate us from nature, we further the consumption and destruction we see all around us. But I think we can see past the empty goal of creating trillion dollar markets for our products. As humans, we've always had the infinite power to change our minds. It's time we tap that power by creating tools that connect us—to each other, to the earth—and enable us to meet the real life-or-death challenges we face on this planet. As programmers and technologists we're in a key position to determine the future by the choices we make every day. I hope each of us can make every keystroke and every mouse click a step toward a sustainable society.

RECOMMENDED READING

- *Alan Turing: The Enigma* by Andrew Hodges (Simon & Schuster, 1983).
- *The Three-Pound Universe* by Judith Hooper and Dick Teresi (Tarcher Press, 1986).
- *Who Needs Donuts?* by Mark Alan Stamaty (The Dial Press, 1973).

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.•

MACINTOSH Q & A

Q *Is it my imagination, or does GetPictInfo return a bit depth of 1 on QuickTime compressed PICT files?*

A Yep! This is what's happening: The Picture Utilities Package doesn't know of the QuickTime Compressed Pixmap opcode (0x8200), so it just skips over the opcode's data; then it finds the PacksBitRect opcode containing the black-and-white pseudo-alert that you get when you draw the picture on a machine that doesn't have QuickTime installed, and GetPictInfo reports back this alert.

Trivia: When QuickTime is installed, it displays the compressed image and then ignores the following PacksBitRect since QuickTime knows it's only the black-and-white alert.

Q *Is it true that if I double-click a document belonging to my application, the application will be launched and will receive an 'odoc' Apple event, but will not receive an 'oapp' event—that is, it will receive either 'odoc' or 'oapp' but not both?*

A Yes, except actually it will receive one of 'oapp', 'odoc', or 'pdoc'. The 'pdoc' will be followed (as the next event) by a 'quit' if the 'pdoc' was the event sent as the application was launched.

This is the *normal* sequence of events, and should be adhered to by everyone who launches applications. However, it isn't enforced by the system or the Finder. It's possible for any application to launch your application with *any* event, since it can stuff anything in the launchAppParameters field of LaunchApplication, as long as it's a valid high-level (not even Apple) event. Launching another application this way would be bad programming, and would break most applications, but you should be aware that someone who doesn't understand event handling may do this to you.

Note that if another application launches your application using LaunchApplication and doesn't specify any high-level event in the launch parameter block, the Finder will automatically supply the 'oapp' event. So, in general, if Apple events and launching have been coded correctly, you'll always receive an 'oapp', 'odoc', or 'pdoc'.

Q *I'm using the Picture Utilities Package to extract the color table from a picture. After getting the color table, I use NewPalette to construct a palette from the color table (usage = tolerant, tolerance = 0). After I do this, the RGB values in the palette don't always exactly match the RGB values in the source color table, causing my program to fail. If I use NewPalette without a source color table, and then use CTab2Palette to copy the colors over (again with usage = tolerant, tolerance = 0), the colors match exactly.*

Kudos to our readers who care enough to ask us terrific and well thought-out questions. The answers are supplied by our teams of technical gurus; our thanks to all. Special thanks to Jim "Im" Beninghaus, Neil Day, Matt Deatherage, Tim Dierks, Steve Falkenburg, C. K. Haun, Dave Hersey, Rich Kubota, Scott Kuechle, Edgar Lee, Jim Luther, Joseph Maurer, Kevin Mellander, Jim Mensch, Guillermo Ortiz, Dave Radcliffe, Greg Robbins, Eric Soldan, Bryan "Stearno" Stearns, Forrest Tanaka, and John Wang for the material in this Q & A column.•

A It turns out that NewPalette doesn't use CTab2Palette, but copies the RGB fields in a strange way that's causing the problems you're seeing. NewPalette copies the high byte in each color table RGB entry into both the high byte and the low byte of the corresponding palette entry. Thus, if the color table entry for red was $F000, it becomes $F0F0. This of course makes no difference to QuickDraw since the low byte isn't displayed, but if your program expects the low byte to match, that's where your problem exists. CTab2Palette is different, in that it doesn't copy the high byte into the low byte unless the pmAnimated bit is set.

The best solution for your code isn't to compare the entire RGB value when comparing colors, but rather to compare the high byte of each RGB component separately. If this isn't possible, the next best solution is for you to use the workaround that you've already discovered with CTab2Palette.

It's unlikely that the Palette Manager is going to change in the future for something like this. In fact, we would almost call it a "feature" since other developers may even depend on it.

Q *My application wants to open other applications and play with the resources therein, like ResEdit, but when it calls OpenResFile on an application, the program gets lost in GetNamedResource. Is there something I'm missing?*

A Your problem stems from the fact that some resources in the application file you're opening with OpenResFile are marked to be preloaded, and so are loaded into memory when the resource fork is opened.

Since most applications have CODE resources marked to be preloaded, this turns into a much bigger problem, because the Segment Loader will treat these preloaded CODE resources as your code resources if you make a between-segment call that triggers a call to LoadSeg while the opened resource file is first in the resource chain. If this happens, you'll begin executing code out of the other application, which will cause your Macintosh to crash and burn.

The solution to this problem is to bracket OpenResFile calls with SetResLoad(FALSE) and SetResLoad(TRUE), and to avoid making between-segment calls when you've got another resource file open that contains CODE resources. This will not only prevent your application's memory from being used by preloaded resources that you don't want, but will also prevent the Segment Loader from jumping into the other application's code. If you need to get CODE resources out of the opened resource file, you can still prevent the Segment Loader problem by calling UseResFile on your application's resource reference number to put your application at the top of the resource chain.

Q *How can our application search for files by label or color, getting the actual string for the label/color field, so that users can select from a menu that looks like what they'd see in the Finder or ResEdit?*

A In the icon utilities there's a call that will get you the RGB color and string for the Finder's labels. Information from the May 1992 revision of Macintosh Technical Note "Drawing Icons the System 7 Way " (formerly #306) is shown below. It includes the glue code for the call in MPW Pascal and C formats.

```
FUNCTION GetLabel (labelNumber: INTEGER; VAR labelColor: RGBColor;
   VAR labelString: Str255): OSErr;
INLINE $303C, $050B, $ABC9;
```

The label number is in the range of 0 to 7, and is available in bits 1–3 of the file's Finder flags (*Inside Macintosh* Volume VI, page 9-36). The call returns the actual color and string used in the Label menu of the Finder and the label's control panel. This information is provided in case you want to include the label text or color when displaying a file's icon in your application.

Q *I'm making an asynchronous low-level File Manager call from inside a completion routine (for example, error := PBxxx(@PB, TRUE);). Occasionally on some machines, the call immediately returns an error in the function result even though everything appears to work correctly. Do I need to worry about the result when I make the call?*

A It sounds as if you're making the mistake of testing the function result of an asynchronous File Manager call (the value of register D0 is returned in the function result). There's no useful information in the function result of an asynchronous call made to the File Manager; the call might not even have been looked at by the File Manager yet. The call's result status is stored only in ioResult after the call completes, or in either D0 or ioResult at the entry to the completion routine. If you're polling to check for the call's completion, ioResult will indicate the call has completed when it's less than or equal to 0.

In general, when making asynchronous I/O calls (reads or writes) there are only two types of function result error that are of any possible consequence: a "driver not open" error (notOpenErr) and a driver reference number error (badUnitErr or unitEmptyErr), which indicate the call wasn't successfully queued by the driver and the ioCompletion routine won't be called. Neither one of these error conditions makes any sense for the File Manager (which isn't a driver); the File Manager will *always* call the completion routine (if any) of a given asynchronous call. Your program should just ignore the function result of an asynchronous low-level File Manager call and leave it up to the completion routine or the routine polling ioResult to check for and handle any errors that may have happened on the call.

Q *Many of the new File Manager calls are just HFSDispatch with new selector codes. How do I check whether a given selector is implemented? An example of a new File Manager call is GetVolParms. Currently I don't check, I just read the result code. It seems to be OK. How should I interpret the response from Gestalt when called with gestaltFSAttr? As I read it, gestaltFullExtFSDispatching tells me that all the calls are available. Are there times that only a few of them are available? PBHGetVolParms seems to be available at all times anyway. Where do I find more info on the workings of HFSDispatch? In general I would appreciate some more info on the compatibility issue.*

A There are two issues. One is that not all versions of the File Manager support all calls. The other is that even if the File Manager supports the calls, individual volumes may not.

The first issue is addressed by Gestalt's gestaltFSAttr selector. Before System 7, HFSDispatch supported a fixed range of selectors. The result was that some advanced file services were unavailable, even though the volume would support them. A good example is PBCatSearch. If you mount an AppleShare 3.0 or FileShare volume under System 6 with the AppleShare 3.0 Chooser extension, the volume will report via PBHGetVolParms that bHasCatSearch is true. But if you try to make the call, you'll get back a paramErr because HFSDispatch doesn't know about the CatSearch selector.

System 7 doesn't restrict the range of HFSDispatch selectors. For external file systems, this means it's up to the external file system to determine whether it can handle the selector and to return an appropriate error if it cannot. This is the meaning of the gestaltFullExtFSDispatching flag. If it's true, there are no limitations on the range of selectors.

The second problem is that even though HFSDispatch won't limit the range of selectors, the volume may still not support the call. To turn the previous example around, making a PBCatSearch call in System 7 to a pre-AppleShare 3.0 volume will result in an error because the volume doesn't support the call.

The best way to determine whether a volume supports a feature is to use PBHGetVolParms. This can return most of the information you need about advanced file system calls. Unfortunately, there can be problems even with that. For example, when the user turns file sharing on and off, the bHasPersonalAccessPrivileges flag can change. So you can't just test attributes once and assume they'll never change.

As far as knowing whether PBHGetVolParms is available, this is not a new call. It's documented in Chapter 21 of *Inside Macintosh* Volume V. The way to check for it is to simply call it and if you get back a paramErr, it's unsupported (page 387). This seems to be what you're doing, so you should be safe.

To summarize, there's no guaranteed way to know if a particular selector will work (but it should never crash, just return paramErr). The right sequence of steps is to first check to see if the HFS supports the full range of calls, then check for specific features using PBHGetVolParms. And in any event, you should always check for errors and be prepared to take appropriate action. A good example of how to do this can be found in the January 1992 version of the Macintosh Technical Note "Searching Volumes—Solutions and Problems" (formerly #68).

Q *How does Developer Technical Support manage to answer so many difficult questions so accurately?*

A We swear by the Magic 8-Ball as a technical reference. Not only is it convenient, user-friendly, and available at your local toy store for less than ten dollars, but it's guaranteed 100% correct. This way, we manage to answer all questions quickly and accurately and still leave time for playing Spaceward Ho!

Q *What's the purpose of the MacApp 'mem!' and 'seg!' resources, and where does the documentation for these resources exist?*

A The 'mem!' resource allows you to change MacApp's memory allocation reserves in various ways. Each contains three numbers: the amount to add to the temporary reserve, which is used for system allocations such as system resources and temporary handles; an amount to add to the permanent reserve, which is used by you for your memory allocation; and an amount of stack space. Having multiple 'mem!' resources causes their values to be summed; in this way, you can create a "debugging" 'mem!' resource that gives you extra space and delete it when you produce a non-debug version. This is discussed in the *MacApp 2.0 General Reference*, in Chapter 3.

The 'seg!' resource is used to reserve space for code segments. If the Macintosh ever tries to load a code segment but fails due to lack of memory, it will crash. Thus, MacApp keeps a store of memory solely for loading code resources. It sizes this reserve by adding together the sizes of the segments named in the 'seg!' resource. One way to do this would be to just name all the segments, so that you know there's room for them all; however, this would be wasteful, because many segments are often unused (your printing code, for example). So what you do is name only those segments that represent the largest code path you can have—the calling chain that would require the largest set of code segments to be loaded at any time. This is also described in Chapter 3 of the *MacApp General Reference*. In contrast, 'res!' names segments that *must* be resident all the time; they're actually loaded and made resident, as opposed to the 'seg!' segments, which are used only to calculate how much memory should be reserved for segments in general.

Q *I've been thinking of shutting down the System 7 Finder. Is this a cool thing to do in my application?*

A We normally recommend that you don't quit the System 7 Finder application. Nevertheless, there may be a few good reasons to shut down the Finder. For example, the Installer (the only application Apple ships with a good reason to do so) sometimes needs to shut down the Finder and all other applications to make sure system resources aren't being used while they're being updated by the Installer.

If you find yourself in a situation where you need to shut down the Finder, you should know about a few things:

- Before you shut down the System 7 Finder, use the Process Manager to see if the File Sharing Extension is running. If so, you should shut it down before shutting down the Finder. The File Sharing Extension shouldn't be running without the Finder because the Finder is the only user interface the File Sharing Extension has. You shouldn't take away the user interface to file sharing.

  There's another good reason to shut down the File Sharing Extension before the Finder. The Network Extension (not the Network control panel) handles all the user interface transactions among the Finder, the File Sharing Monitor control panel, the Sharing Setup control panel, the Users & Groups control panel, and the File Sharing Extension (the file server). The Network Extension opens another file, the Users & Groups Data File, so that it can manipulate users and groups. When you shut down the Finder (with a kAEQuitApplication Apple event), the Network Extension and its connection to the Users & Groups Data File are also closed (almost). Because of a minor bug in the system, the File Manager thinks that the file is closed and that the FCB used by that access path is free for reuse; however, the File Sharing Extension thinks the access path to the Users & Groups Data File from the Network Extension is still open. When the File Manager attempts to reuse that FCB to open another file later, the file is opened, but because the File Sharing Extension thinks that FCB is still in use by the Network Extension, it won't allow access to the file and it returns opWrErr (-49) to the Open call. At this point, the file that someone was attempting to open can't be accessed or closed.

- If the Finder is shut down and then eventually relaunched, there may be some fragmentation of the MultiFinder heap. This can occur because the Finder is the first application to be started, so it's always first in the MultiFinder heap. When you shut it down, that memory becomes available and other processes might occupy that space. When the Finder is restarted, if it can't get into its original space in the MultiFinder heap, it will get loaded somewhere else and probably won't be shut down again.

- In System 7, the Finder is responsible for filling the Apple menu with the items in the Apple Menu Items folder. When the Finder is gone, so are the Apple menu items, including things that are important to most users (like control panels).
- If the user has selected background printing with a LaserWriter or StyleWriter, nothing will print while the Finder is gone. That's because the Finder is responsible for monitoring the PrintMonitor Documents folder and launching the PrintMonitor application when necessary.

Q *My Balloon Help message doesn't appear when I use a 'TEXT' resource in a static window as the message in the balloon, following string resource examples in Inside Macintosh Volume VI and modifying the code to indicate 'TEXT'. Why doesn't this work?*

A While using 'TEXT' resources in Help Manager balloons is a way to provide stylized text, it doesn't mean that strings longer than 256 are possible. In fact, strings up to only 239 characters in length are valid. When string lengths are greater than 239 the Help Manager takes a short-circuit return and no balloon is displayed. To work around this limit, you can draw a picture with the text you want to display and then use the picture in the Help Manager balloon.

Q *My TrueType font has all 256 characters defined with a unique glyph. I've been unable to draw the $20 (space) character. DrawChar, DrawText, DrawString, and DrawJust all ignore this character in the font and draw a blank character. How can I draw it?*

A Unfortunately, the problem with the space character not being drawn is hard-coded into the text-drawing routine in the core of QuickDraw. ASCII 32 is always "optimized away," regardless of the font being used or of the particular circumstances. The only workaround is to put the corresponding character elsewhere in the ASCII character encoding (or, if this isn't possible, to use an additional font).

You're lucky that TrueType fonts always render the ASCII code 13 (carriage return) if it has a glyph in the font; for bitmapped fonts, if the character drawing happens with scaling, or with foreground/background colors different from black/white, even the CHR(13) drawing is optimized away.

Q *After connecting to a remote network via AppleTalk Remote Access (ARA) I can call PGetAppleTalkInfo and it returns the proper zone name. However, after disconnecting, PGetAppleTalkInfo still returns the remote network's zone name instead of "*" or nothing as I would expect. Is there some period of time I should wait before expecting the zone name and network number to return to zero (no internet)?*

A This is, in fact, a bug with ARA version 1.0. Apple is investigating the problem and there will be a fix in a future release. An easy workaround is to check the GetZoneList or GetMyZone call to see if it returns any zones.

Q *My application opens a number of resource files. If my 'hrct' resources aren't in the most recently opened resource file, I get an error -192 (resource not found) from HMGetIndHelpMsg. Is this a Help Manager bug?*

A Your 'hrct' resources get loaded only from the most recently opened resource file because that's where the Help Manager is looking for them. The Help Manager uses Get1Resource to get 'hrct'-type resources to avoid Resource Manager conflicts with other resources. This is simply a behavior of the Help Manager. To use HMGetIndHelpMsg properly, call UseResFile with the refNum of the file containing the 'hrct' before you call the Help Manager. If the 'hrct' is stored in your application resource fork, you can use HomeResFile or CurResFile (at the start of your program before you've opened any other resource files) to get the refNum of the application resource file.

Not all Help Manager resources are treated this way, due to the design of the Macintosh Toolbox; 'hdlg' resources, for example, are loaded using GetResource so that things like Standard File can have common help throughout all applications.

Q *We're having problems with the GetScrap function in our desk accessory. After a user opens our DA, when we call GetScrap to get any text from the Clipboard, GetScrap returns -102 (no requested type in scrap). After once (or more?) through the event or SystemTask loop, the scrap suddenly shows up. What's causing this?*

A The reason for the trouble you're having with desk accessories and GetScrap is that you're looking for the converted scrap at the wrong time in the process. According to the MultiFinder documentation (*Programmer's Guide to MultiFinder* and *Inside Macintosh* Volume VI), an application is supposed to convert its private scrap and write it to the public scrap when it receives a suspend event. When the system opens your DA, it immediately sends it an Open message; the application hasn't received its suspend event yet. You have to wait until the scrap has been converted. This, for a DA, should occur at the first null event, and for applications, when you get a resume event.

Q *QuickTime is a joy! But I've run aground with SetMovieRate. I'm trying to change the rate at which a movie plays back, but if I call SetMovieRate the movie starts playing immediately, the controller goes wild, and the next time I hit the play button it ignores the previous rate. How can I control my playback rate?*

A SetMovieRate takes effect immediately; that's why the movie starts playing as soon as you make the call with a rate other than zero. Also, calling SetMovieRate behind the controller's back can only cause confusion because you're changing the state of the movie without letting the movie controller know about the change. Note that in normal operation the movie controller plays back movies at the standard speed, rate = 1; this is the current behavior. It's possible that in a future release the movie controller will use the rate the movie was saved with or the one set with SetPreferredMovieRate.

A little-known fact is that the standard controller does contain a primitive mechanism for controlling the rate of playback. If you hold the Control key down and then click the stepping buttons, you can, for example, play the movie backward. Furthermore, if you hold the mouse button down you'll get a slider control that does let you play the movie at different rates backward or forward.

The slider provided by the standard controller isn't intended to set the rate, so if you play once at low speed the rate doesn't stick and, as you've found, the next time you click the play button you go back to the normal speed. If you need the selected rate to remain for the session, you'll have to provide your own method of selection.

Once you know your desired speed, you'll need to provide your own filter procedure and install it calling MCSetActionFilter. Upon receiving any mcActionPlay actions for rate changes, you'll need to call SetMovieRate to set the movie in motion at the desired rate (and return TRUE). Using a filter procedure is the proper way of doing this because the controller can keep in sync with the actions even though it's your code that actually affects the action.

Note that you'll have to do some extra work to mimic the normal behavior of the standard controller. For example, when you're at the end of the movie and the user hits the play button, the controller goes back to the beginning and plays the movie. Your filter proc has to do the same when playing back the movie at a rate different from the normal. A different behavior will confuse the user.

For details on filter procedures, controller actions, and the movie controller in general, see the "QuickTime Components" chapter of the *QuickTime Developer's Guide*.

Q *How does the Magic 8-Ball achieve its high level of technical accuracy?*

A There are two theories on this: The first is that the Magic 8-Ball picks an answer at random and then alters reality to fit the answer it has picked. For example, if you were to ask it "Am I a millionaire?" it might pick "Signs point to

yes" at random. Then it would either have to warp reality so that you actually would have a million dollars or, more likely, warp your memory of the question so that you would think you'd asked "Is Elvis still alive?" The other theory is that it's just magic.

Q *I'm having trouble with PBGetCatInfo returning old data to my application. For example, if I change a file label using the Finder Label menu and then run my program, which calls PBGetCatInfo, the fdFlags field in the FInfo record returned doesn't reflect the change. I've tried calling PBFlushVol before I do this; the file isn't open, so there's no way to call PBFlushFile. However, restarting the Macintosh or changing the file's name causes PBGetCatInfo to work correctly. What's going on and how do I get around it?*

A The Finder caches much of the "Finder information," including things like the color coding information users can set with the Finder Label menu and the view position of objects in folder windows. Changes to the Finder information are cached until the folder that contains the objects that were changed is closed (which happens at system restart or shutdown time) or until some noncached change is made to the object (for example, the file is renamed). The Finder caches what it considers Finder-specific information to cut down on the number of disk accesses it must make. (For example, rearranging the object view in a window would be very slow on floppy disks if the Finder wrote to the disk every time the user drags a group of objects around.) Since in most cases no other applications should care about the state of the Finder information, this normally doesn't cause problems. There's no workaround for this behavior in the current implementation of the Finder.

Q *The Macintosh QuickDraw routine ObscureCursor hides the cursor until the next time the mouse is moved, but it isn't affected by HideCursor or ShowCursor. Our application needs to use ObscureCursor while the user is typing but needs the cursor to be visible after no typing has occurred for a short period. How do we "undo" ObscureCursor, since we can't rely on the user to move the mouse?*

A The only way (besides actual mouse movement) to make an obscured cursor visible again is to convince the system that the mouse has moved. There's no really good way to do this via Toolbox calls, so you're going to have to do it the hard way and simply update the low-memory cursor information to tell the system the cursor moved (even though you don't need to update the actual position).

To tell the system the cursor has changed location, simply set the crsrNew flag (a byte located at $08CE) to 1. When the system sees this byte is 1, it will assume the cursor has moved and redraw the unobscured cursor at the

appropriate place (where it was all along), and reset CrsrNew, waiting for the mouse to move again.

Q *In System 7, I want to place my user's preferences file in the Preferences folder in the System Folder, but I can't seem to get the Preferences folder's directory ID and other information so that my file will appear there! Also, how do I get to that folder if the user changes the names of the System Folder and Preferences folder? And once the user's preferences file is there, am I assuming correctly that the best way to find it again is to make an alias record to track the file ID?*

A System 7 introduced the routine FindFolder for locating the Preferences folder. Just make this call:

```
err := FindFolder (kOnSystemDisk, kPreferencesFolderType,
   kCreateFolder, prefVRefNum, prefDirID);
```

If FindFolder returns noErr, prefVRefNum and prefDirID will contain the vRefNum and dirID of the Preferences folder, which can be used later with HCreateResFile, HOpenResFile, PBHGetFInfo, and other File Manager calls to locate your preferences file. If a Preferences folder doesn't already exist, the kCreateFolder parameter instructs FindFolder to make one and return the vRefNum and dirID of the new folder.

FindFolder is documented in Chapter 9 of *Inside Macintosh* Volume VI, under "The System Folder and Its Related Directories." Although FindFolder is implemented only in System 7, if you're using MPW 3.2 (or the current THINK compilers) glue is automatically included in your compiled code, making it safe to call FindFolder in System 6. The glue checks whether FindFolder is available and, if it isn't, returns the System Folder's vRefNum and dirID for the kPreferencesFolderType selector. Use the System Folder values as the location for the preferences file in System 6.

If you're not using a development system that provides the FindFolder glue, your code should check the FindFolder Gestalt selector gestaltFindFolderAttr to see if FindFolder is available. If FindFolder is available, call it. FindFolder is defined as

```
FUNCTION FindFolder (vRefNum: INTEGER; folderType: OSType;
   createFolder: BOOLEAN; VAR foundVRefNum: INTEGER;
   VAR foundDirID: LONGINT): OSErr;
INLINE $7000, $A823;
```

If FindFolder isn't available, call SysEnvirons to find the System Folder's working directory reference number, call PBGetWDInfo or GetWDInfo to

convert that number to a true vRefNum and dirID, and use those System Folder numbers for the location of the preferences file. Example code for this is in the Q&A stack, under Operating System:File System:Code for identifying vRefNum and dirID of Macintosh System Folder.

To locate the Preferences folder, follow the steps described above rather than trying to keep an alias of the Preferences folder or of the preferences file. However, if there are any other files in the System Folder that the application depends on (such as dictionaries) those should be tracked with aliases, stored as 'alis' resources in the preferences file. See Chapter 27 of *Inside Macintosh* Volume VI for information on using aliases.

Q *I recall reading that QuickTime includes an implementation of the Alias Manager for System 6, but I haven't found any precise description of what's included. Is it a bare minimum to support QuickTime? Or is the full Alias Manager there? Also, is there any way I can use the FSSpec interface to the File Manager, or must I revert to the System 6 interface?*

A When QuickTime is installed, most of the Alias Manager is available in System 6, with these exceptions:

- NewAlias will accept a fromFile parameter, but it never creates a relative alias.
- NewAliasMinimalFromFullPath and ResolveAliasFile aren't available.
- ResolveAlias and UpdateAlias will accept a fromFile parameter, but ignore it.
- MatchAlias may be called, but the kARMMultVols, kARMSearchMore, and kARMSearchRelFirst flags aren't available. If you pass them in, they'll be ignored. Furthermore, if you pass in a matchProc, it will never be called.
- The System 6 Alias Manager won't mount network volumes.

To summarize, in System 6 the Alias Manager doesn't handle relative aliases, multiple volume searches, "searchMore" searches, and network volume mounting. On the bright side, nearly all calls are present. Aliases created in System 6 are compatible with System 7 aliases, and aliases created in System 7 will work in System 6.

Unfortunately, QuickTime doesn't currently install an Alias Manager Gestalt selector, since it's only a partial implementation. You can check for the Alias Manager using Gestalt and, if it isn't present, look for QuickTime (using Gestalt); if QuickTime is present, assume you have an Alias Manager, subject to the limitations listed above.

QuickTime also makes extensive use of the FSSpec data structure introduced in the System 7 File Manager. Nearly all the FSSpec calls are available in System 6 when QuickTime is installed. The following calls are available in System 6, and should behave as documented for System 7: FSMakeFSSpec, FSpOpenDF, FSpOpenRF, FSpCreate, FSpDirCreate, FSpDelete, FSpGetFInfo, FSpSetFInfo, FSpSetFLock, FSpRstFLock, FSpRename, FSpCatMove, FSpOpenResFile, FSpCreateResFile, and FSpGetCatInfo. FSpExchangeFiles isn't available when using the QuickTime System 6 version of the FSSpec calls.

Again, the Gestalt selector for the FSSpec calls isn't installed when QuickTime is there. This means that the gestaltFSAttr Gestalt selector may not be present, and gestaltHasFSSpecCalls may not be set, even if gestaltFSAttr is present.

Q *I would like to implement the preview/thumbnail feature in the Standard File dialog, just like the extension included with QuickTime. Is that code available separate from QuickTime? If not, could I at least get information on how the preview is created?*

A To implement your own preview/thumbnail feature, simply duplicate the Standard File dialog, add the necessary 'DITL' resources, and install a custom filter procedure for handling preview commands. On the System 7 CD there's an example, StdFileSample, that shows exactly how to create a custom file dialog. The Macintosh Technical Note "Customizing Standard File" (formerly #47) describes how to do this as well. For generating and displaying the preview, you can use the following PreviewResourceRecord, found at the end of the ImageCompression.h file:

```
struct PreviewResourceRecord {
   unsigned long     modDate;
   short             version;
   OSType            resType;
   short             resID;
};

typedef struct PreviewResourceRecord PreviewResourceRecord;
typedef PreviewResourceRecord *PreviewResourcePtr, **PreviewResource;
```

This is the format for the 'pnot' resource, which defines the preview for the movie file, usually pointing to a 'PICT' resource. It's all you need to generate QuickTime-compatible preview without using QuickTime.

Q *You guys don't really use the Magic 8-Ball to answer programming questions, do you?*

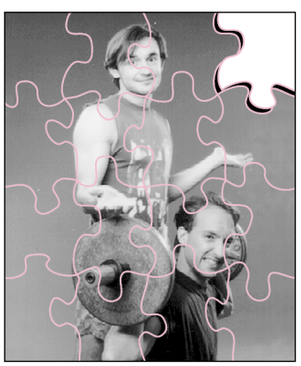
A Reply hazy, try again.

Have more questions? Need more answers? Take a look at the Dev Tech Answers library on AppleLink (updated weekly) or at the Q & As on the *Developer CD Series* disc, or ask a Magic 8-Ball.•

KON & BAL'S PUZZLE PAGE

AN OFF-COLOR PUZZLE

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

100	BAL	This guy has this program that leaves all the palette colors messed up after his program quits.
	KON	Yeah, I've seen it. He's probably got version 1.0 of that KeithPaint program.
90	BAL	This is something he wrote with MPW C, and he spent days debugging his code with SADE. He's sure it's not his problem.
	KON	Sounds like some kind of Palette Manager nastiness.
80	BAL	He sends you a copy and it works fine on your machine.
	KON	Which system is he running? How many monitors does he have?
75	BAL	You're both running 6.0.7 on a Macintosh fx with one Apple 13-inch color monitor.
	KON	That's a nice programming environment. Why doesn't he upgrade to System 7 and buy himself a real monitor?
70	BAL	Well, he has System 7 on a Macintosh Quadra with a 16-inch color monitor for reading NetNews and watching QuickTime movies. But the bug doesn't happen on that machine.
	KON	So it must be some INIT conflict. What's he got, ColorDesk or something?
65	BAL	Nope, happens without any INITs.

KONSTANTIN OTHMER AND BRUCE LEAK
Between vacations, KON and BAL often find themselves under a lot of pressure to catch up on their work load. When BAL gets himself in too deep, he calls KON in for assistance. They try to hash out their puzzling problems at Apple's Fitness Center—although KON admits to having trouble staying focused, due to what we can only call his "wandering eye." When not working on Puzzle Pages and *develop* articles, KON and BAL actually write software. As a follow-on to their previous QuickDraw and QuickTime successes, look for QuickFit, featuring QuickTime-based exercise videos and QuickBuf protein powder. You read it here first!•

KON He's using the Color Manager and calling some nasty thing like SaveEntries or RestoreEntries, and he isn't MultiFinder compatible. I tell him to use the Palette Manager.

60 BAL He's using the Palette Manager, and he's totally MultiFinder friendly.

KON Sounds impossible. I swap motherboards with him.

55 BAL Still happens, just on his machine. If you swapped hard drives it would happen on yours.

KON It's got to be some kind of GDevice color table thing. I check the GDevice color table before and after running his program.

BAL Wait a second. Who has which hard drive?

KON I have mine, he has his.

50 BAL The color tables are different on his GDevice, but the same on yours.

KON Well, who restores the color table? PaletteMgrExit or something like that, right?

45 BAL The Palette Manager changes the color environment only when a window with an associated palette comes to the front. This method assumes that people who need colors request them, and people who don't request colors don't care enough to affect the color environment.

Unfortunately, with this approach to color management, when an application that's Palette Manager intensive quits, and an application such as the Finder which has no intensive color demands comes to the front, you're stuck with the nonstandard color state of the application that just quit.

In 32-Bit QuickDraw David Van Brink extended the Palette Manager to include a routine called PaletteMgrExit. This routine is called automatically for you when your application quits, thus restoring the default color state.

KON Don't you have the same problem when twitching between layers? If you twitch to the Finder and the other application is still running, you'll be stuck with the nonstandard color state too.

40 BAL Yeah, but there's still an application around that needs those colors, and you can potentially see that application's documents, so it's a good bet to keep them. No one is specifically asking for the standard colors, and someone wants the others, so why not keep them around? If you twitch to another application that had palettes, obviously its color needs will be satisfied.

KON So in effect you start out with a default color world, and you launch an application that twists the colors to the demands of a particular

SCORING

90–100 So you're still using System 6 with SADE? Upgrade to System 7 immediately!
60–80 Do you swear to tell the truth, the whole truth, and nothing but the truth?
35–55 Good job. Next time you're in L.A., try out for "Jeopardy."
5–30 Hey, we're with you! MacsBug users through and through.•

document you're viewing or editing. Upon quitting that application your machine is left in some nonstandard color state. So PaletteMgrExit cleans up the mess.

35 BAL That's basically it, but just going back to the default color environment when an application quits isn't sufficient. PaletteMgrExit reverts to the default color set modified to accommodate the application coming to the front.

KON I set an ATB on PaletteMgrExit and see if it gets called.

30 BAL It's called on your machine, but not his.

KON Well, ExitToShell is supposed to call it, and it's hard to call ExitToShell wrong, so there must be some system problem.

25 BAL Welcome to the Puzzle Page, KON. Maybe this would be an excellent opportunity to fire up one of those many fine Macintosh debugging environments, Mr. MacsBug.

KON Clearly _ExitToShell is different. So I list ExitToShell with MacsBug and see who's there.

20 BAL It's in RAM.

KON I see whose heap it's in using HZ.

15 BAL It's above all heaps.

KON So it's in MultiFinder memory. He must be running an old version of MultiFinder that doesn't know about PMgrExit.

10 BAL Yeah, you know that SetAside MultiFinder they still ship with SADE?

KON Version 6.1b9. Sounds like they never really finished it.

5 BAL Well, my hero Phil Goldman broke off the sources to add that SetAside stuff, but it was never shipped as an official Apple release, since System 7 was just around the corner. SADE needed that feature because they twitch in some weird way. So they're sort of stuck using that version, while the rest of the world moves on. And they keep shipping that old MultiFinder for those last two, die-hard, System 6 developers. System 7 SADE users don't have this problem.

KON Nasty.

BAL Yeah.

Thanks to Sean Callahan, Scott Douglass, and David Van Brink for reviewing this column.•

INDEX

For a cumulative index to all issues of *develop*, see the *Developer CD Series* disc.•

Q

R

S

T, U

V

W, X, Y

Z