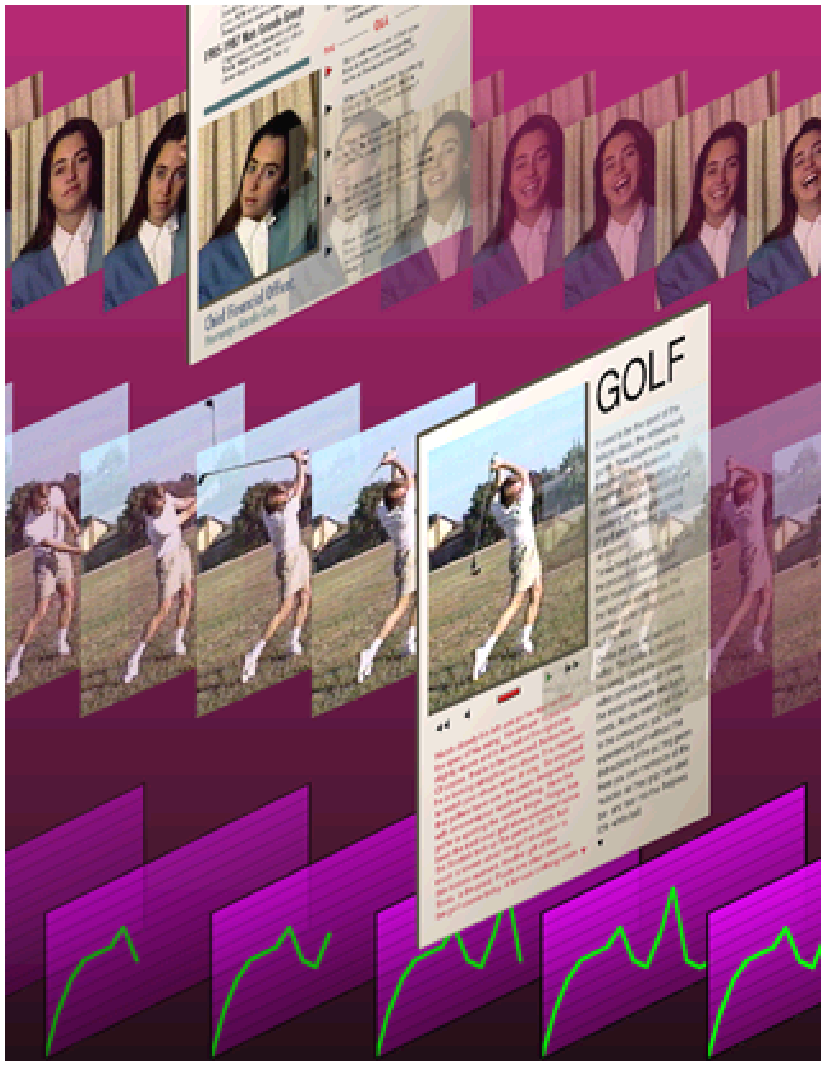
develop

The Apple Technical Journal



Issue 7 Summer 1991
Apple Computer, Inc.

EDITORIAL

Editor-in-Cheek *Caroline Rose*

Spirited Guide *Louella Pizzuti*

Technical Buckstopper *Dave Johnson*

Managing Editor *Monica Meffert*

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

Editorial Assistant *Helen Stea*

Indexer *Ira Kleinberg*

Manager, Developer Support Systems and Communications *David Krathwohl*

ART & PRODUCTION

Production Manager *Hartley Lesser*

Art Director *Diane Wilcox*

Technical Illustration *Automatrix, Don Donoughe, and John Ryan*

Formatting *Automatrix*

Printing *Wolfer Printing Company, Inc.*

Film Preparation *Aptos Post Inc.*

Production *PrePress Assembly*

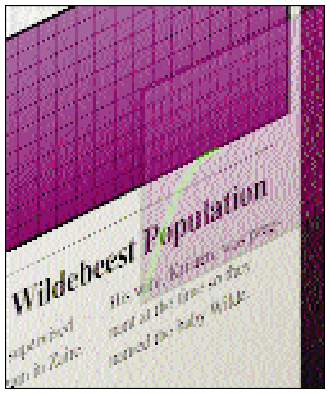
Photography *Sharon Beales, Ralph Portillo, and Steven C. Johnson*

Circulation Management *David Wilson*

Online Production *Cassi Carpenter*

REVIEW BOARD

Pete Alexander, C. K. Haun, Larry Rosenstein, Andy Shebanow, and Gregg Williams



The cover was created by Hal Rucker and Cleo Huggins using Adobe Photoshop™, a RasterOps™ Frame Grabber, and their friends Denise Huajardo and Brian Crowley. Wildebeests were inspired by a documentary on an African watering hole.

d e v e l o p, *The Apple Technical Journal*, is a quarterly publication of the Developer Support Systems and Communications group.

With TrueType, Apple's outline font technology, "Hello, World" has never looked so good. See "Scoring Points With TrueType" for details.

CONTENTS

CAROLINE ROSE

Dear Readers,

My editorial in Issue 6 ended with this riddle: "I entered this entire editorial without pressing a single key on the keyboard or clicking the mouse button. I was as quiet as a mouse (the furry kind). How did I do this? And furthermore, why?" Well, first I had an on-screen keyboard (a desk accessory) that interprets a click on one of its keys as a press of that key. This DA has an option that lets you set a delay after which the mere presence of the cursor over that key will be interpreted as a keypress. I also had a trackball set up on the floor and used my foot to move the cursor around. So with this I was able to type without using my hands.

As for why I wasn't using my hands—besides as a way of getting a snappy ending to the editorial—I'm one of many people who suffer from RSI, repetitive strain injury. In my case, this means tendinitis in my forearms, but it can also mean carpal tunnel syndrome and a host of other similar problems. Since my recent return to Apple®, I've learned that there are many software and hardware products for the Macintosh® that can help RSI sufferers and others with limited hand movement. I'm now using a trackball with my nondominant (less-suffering) hand and with a foot switch—a pedal I step on to click. I'm also using a desk accessory that alerts me when I've been typing for a half hour without at least a five-minute break (I chose delicate Tibetan-style flute music as my auditory cue, but there are of course less sublime options). There are many similar products that I haven't yet explored. With diligent stretching and breaks from typing—and freedom from using the mouse—I'm able to type as much as I need to in order to do my job. Many others are less fortunate.

That's the good news. The bad news is that many of these products don't work with other, mainstream software: the on-screen keyboard isn't compatible with a certain macro program I'd also benefit from using, or with the word processor I use most of the time. This is a very real example of the effect of programming things in nonstandard ways—for example, using GetKeys to find out what characters have been typed rather than getting this information out of the event record, or calling the ADB Manager when you're not writing a special driver and so really should be using higher-level routines. (You know who you are.)

Incompatibilities notwithstanding, these products are terrific, and thanks go to all the Macintosh developers who have created them. You've made a big difference in some people's lives—probably including your own, since I've learned that the

CAROLINE ROSE has been writing computer documentation ever since "timesharing" meant mainframes, not condos. After a seven-year digression into programming, she returned to writing and joined Apple to document the inner workings of a beguiling new computer named Macintosh. The result was a three-volume tome that was affectionately nicknamed "The Vault of Horror." In what proved to be another digression, she left Apple to launch NeXT's documentation effort (starting, interestingly enough, with writing the WriteNow™ For Macintosh manual). She's thrilled to be back at Apple with all its charms. Caroline is an avid reader, swimmer, dancer, and hiker, and is passionate about her cat and all things Italian. Seeing Michael Crawford in *Phantom of the Opera* was a recent high that she's not sure how she'll top (but she'll try).•

motivation for many of you has been that you've had a repetitive strain injury yourselves. For those of you who haven't yet had the problem, you'd be wise to takes steps toward prevention. Don't ignore it; if you slave over the keyboard for long hours, it will probably not ignore you.

While we're on the subject of doing the right thing, I might add that *develop*'s paper is now recycled enough to pass California's stringent requirements for the use of the familiar recycle logo (which we now proudly bear on our back cover). Formerly we used paper that was 50 percent de-inked (waste paper from printing plants, with the chemical inks removed); now our paper is also 10 percent post-consumer waste (not de-inked). Recycled paper keeps getting better looking and more practical to use; we're happy to be able to do our part toward saving the forests. Please do yours, and recycle your issues of *develop* if you don't want to hold on to them—preferably by passing them on to a friend!

As always, we welcome your comments and suggestions. Keep those cards and letters coming . . .

CRose

Caroline Rose
Editor

SUBSCRIPTION INFORMATION
Use the order form on the last page of this issue to subscribe to *develop*™. 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 reverse of the order form on the last page of this issue. Note that back issues are also on the *Developer Essentials* CD-ROM disc.•

LETTERS

NEAT THREADS

I've just finished experimentally adding the Threads Package to my current application. I was amazed at how painlessly it could be added. What I felt was truly remarkable was that I use THINK C's™ object-oriented extensions (which I refer to as C+-) along with the THINK Class Library, and it took less than 10 minutes to perform all the necessary conversions. Pretty neat.

I'm linking to ask whether you have any information about using the Threads Package with THINK C and TCL, and whether there are any special considerations involved in using threads with or inside methods. Just using them inside a method with a single instance, with no reference to instance variables, is demonstrably effective, but this case is logically indistinguishable from conventional code. I suspect that use with objects might require custom fSwapIn and fCopyContext routines to preserve **this,** the handle to the object's instance. In THINK C, **this** is internally kept in an address register.

If the default context-saving routines save A0-A4 register states as well as the stack, **this** should automatically be preserved. Whether registers are preserved does not seem to be documented in your otherwise nifty *develop* article. Logic says some must be preserved (A5 at least), but have you been prescient enough to save all of them?

—Kirk Kerekes

Thanks for your truly inspiring link! It really makes a difference for me to get feedback like this.

I don't foresee any special problems threading THINK C code. All the data and address registers are saved. The FPU registers are saved only if you specifically request that they be saved at InitThreads time, and then only when you have an FPU. Remember to be careful about segment unloading.

I'm at your service if you need any help with threads. Feel free to contact me by telephone. My number is (408)974-0355.

—Michael Gough

MISSING SNIPPETS

Am I blind? Issue 6 of *develop*, page 88, talks about code snippets, but I can't find them anywhere. Are they inside a stack somewhere, or did they miss getting on the CD?

—Greg Johnson

You are not blind. Snippets did not make it onto that CD, but they've made it onto the Developer Essentials disc for this issue of develop. *They're also available via AppleLink®, in the Developer Technical Support folder on the Developer Services Bulletin Board, as well as in the Dev Tech Answers library.*

—Caroline Rose

MISSING TRUETYPE INIT

Recently I received Issue 6 of *develop*. I enjoy reading the articles and would like to make a comment.

Since Apple is distributing a TrueType™ INIT for System 6.0.7, why didn't you put it on the CD of Issue 6? I hope I can find it in the next issue even though System 7.0 is now available.

—Tetsuya Ishikawa

COMMENTS

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 name and company name as well as address and phone number. Letters may be excerpted or edited for clarity (or to make them look like they say what we wish they did).

I hope we can fulfill other people's wishes as easily as we did yours. The TrueType INIT for System 6.0.7 is now on the CD in the folder with the System; an oversight kept it off of the last CD.

As with all old System software, we're providing this INIT so you can test your software with it (just in case you've got some as-yet-unupgraded users). When testing with the TrueType INIT, make sure you use it only with System 6.0.7; that's the only System it's designed to work with.

Happy testing,

—Caroline Rose

DISAPPOINTING CD

I must admit to being disappointed with the CD-ROM disc that came with Issue 6: Tech Notes "stuck" back in 12/90, no Volume VI of *Inside Macintosh*, just HyperCard® alone rather than a developer's edition, no System 7.0. Should I expect that my perception that the disc is out-of-touch, out-of-date, and insufficient will be permanent? That is, *develop* is not really meant to be a real developer-support package for the individual (noncorporate) developer operating on a shoe string? Thanks for any insight you can give me.

—Pete Roberts

I certainly hope your perception that the disc is out-of-touch, out-of-date, and insufficient will not be permanent. We collect and press as much as we can, but because we want subscribers to get develop *regularly, we don't hold the presses for software or documentation that isn't quite ready yet, as was the case last time for System 7.0 and Inside Macintosh Volume VI. They're both on this issue's disc.*

The Tech Notes on Issue 6's disc were actually updated through February 1991. (Well, the stack version was, anyway; the MacWrite® version wasn't, and we apologize for the oversight.)

As for the developer edition of HyperCard, Claris Corporation no longer allows us to distribute it.

We'll continue to do our best to give you the latest, greatest information possible!

—Caroline Rose

WHERE'S LOUELLA?

Congratulations, Caroline, on your new job as Editor-in-Cheek of *develop*. I hope you have as much fun at it as Louella had.

Thanks for the riddle at the end of your first editorial. We've been scratching our heads over it for a while. The closest thing to a guess we can come up with is that you used either a tablet or some other alternative text entry device. You probably were able to emulate the mouse as well as the keyboard. Why? Perhaps to demonstrate that handicapped people can have access to the Macintosh.

A humble suggestion: I've saved all the CD-ROMs that have come with your magazine (*develop* the CD, aka *develop*, the disc, aka *Developer Essentials*). How about designing inserts for the CD cases that some of us keep their CD-ROMs in? Something we could print on thick paper, cut out, and stick in the plastic boxes to label the contents.

Keep up the good work!

—Lyle D. Gunderson

P.S. I tried to send mail to Louella at pizzuti1@applelink.apple.com, but your system denied knowing about her. Any help you could give me in addressing e-mail to her would be very much appreciated.

Thanks for the nice letter. I'm having more fun than I've ever had on a job.

Regarding the answer to my riddle: If you've read this issue's editorial, you know by now that you were barking up the right tree. Other readers who replied did not think of access by people with physical limitations. I hope I've succeeded in doing some consciousness raising here.

Your idea about the insert is a good one. We didn't manage to get it onto this issue's disc, but we'll try for next time. It sounds as if you're holding on to all the old discs. If so, be careful about using stuff on them, because we update software and generally attempt to correct the mistakes of the past with each new disc.

Louella decided that there was after all no job as much fun as being editor of develop, *so she retired to raise flowers in Holland. Just kidding. You can reach her at louella@applelink.apple.com.*

—Caroline Rose

POLES AND FONTS

Two things I'd like to mention after reading Issue 5 (Volume 2, Issue 1):

First, I'm not sure that the answer to the question "What is the difference between North and West?" is completely correct. To my mind, there are two points on the globe from which one would be hard pressed to go further West (or East): the North and South Poles.

Second, may I make a typographic recommendation? Please use Courier for "computer voice" (program listings and the like). Prestige Elite is ugly and too lightweight. The Bitstream® Courier family has a good regular weight and a bold that would be compatible with your Futura headings.

Oh, and without being too dogmatic, I disagree that a spacious layout is necessarily more effective or attractive.

—Toby Thain

You're right, we mistakenly left out the South Pole. Those PR folks for the North Pole do a really good job at making you forget that the other pole exists at all (especially in December, and that was after all our Winter issue). Thanks for the correction.

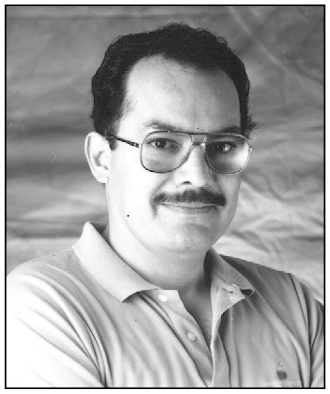
I'm not sure I agree that Prestige Elite is ugly, but it does bother me that its hyphen (what's typed for a minus sign in code) is so narrow and its O (Oh) and 0 (zero) are not very easily distinguishable. So in this issue we have indeed switched to Bitstream Courier, which solves these problems and has the right weight.

The beauty of a particular font or layout is surely in the eye of the beholder; I've heard at least as many positive responses as negative ones to the choices we've made for develop. *We hope that you and others will keep giving us specific feedback so we'll know what's working and what isn't.*

—Caroline Rose

QUICKTIME 1.0: "YOU OUGHTA BE IN PICTURES"

QuickTime is Apple's new architecture for enabling the Macintosh to handle time-based data. The recently introduced QuickTime 1.0 makes it easy for you to add dynamic media like video and sound into your applications—and that's just the beginning. Two sample programs show you how to do the two most basic (and important) QuickTime tasks: playing existing movies and creating new ones.



GUILLERMO A. ORTIZ

The world isn't standing still—it's moving, *fast*—and Apple intends to stay at the front of the race. When Apple came out with the Lisa® and then the Macintosh, the idea of a document file that mimicked a piece of paper was a big deal. But now it's not. Users have taken the "paper" metaphor for granted and are now looking for new metaphors that increase their ability to communicate. One way to do that is to allow documents and applications to contain and display data that *changes*.

QuickTime™ is more than just the ability to record and play back movies—it's a fundamental addition to the Macintosh Operating System. Just as QuickDraw™ gave the original Macintosh the edge of sophisticated graphics in 1984, QuickTime will give the Macintosh another edge over other computers: the *built-in* ability to handle data that changes with time.

Until now, the Macintosh Toolbox has not provided a standard way of dealing with dynamic media, and some developers have come up with their own solutions, especially in the areas of video and sound. Unfortunately, this has resulted in confusion and a lack of standards and basic system support for these data types. Apple has created QuickTime to provide a standard way of dealing with data that changes with time. Even more important, QuickTime gives you the necessary support software so that you can spend your time *using* new data types instead of designing, implementing, and maintaining them.

QuickTime 1.0 works on all color-capable Macintosh computers running Color QuickDraw (models with either a 68020 or 68030 processor) and either System 6.0.7 or System 7.0; a later version will add QuickTime support for monochrome, 68000-

GUILLERMO ORTIZ Instead of giving you details about his life, Guillermo would like to share with you a passage from a soon to be published book. Some say that his affinity for this book tells you much more about Guillermo than we ever could.

Tired, hungry, and thirsty after a long and arduous trek, DunKennsan, also known after his conversion as "He who seeks the Light," entered Brucewhandra's cave, and, without waiting for the religious man to acknowledge his presence, he posed the question burning in his mind: "Why 'QuickTime'?" As a response, Brucewhandra, the man called the Wisest, kept repeating the mantra now famous among the true followers: "Calls that take a Movie can take a Track or a Media. Calls that take a Track can take a Media. Calls that

based Macintoshes. As a result, you, the developer, can take it for granted that QuickTime will be available on *any* Macintosh running your software.

QuickTime 1.0 makes it possible for your program to manipulate the audio/video sequences we call *movies*. (The size, duration, and quality of the average movie largely depends on how much disk space you have for movie files.) It also includes routines for the compression and decompression of still and dynamic images (which should encourage you to use color images without worrying about how much space they take up).

The result of all this is something that you'll like very much: applications and documents that give users a richer experience with your product than they can get with non-QuickTime Macintosh applications or applications on other platforms. QuickTime will make possible a new generation of Macintosh software and hardware solutions that until now have been available only using expensive and narrow-purpose hardware.

ENOUGH MARKETING STUFF! NOW THE DETAILS . . .

QuickTime 1.0 contains the following parts:

- *Movie Toolbox*. This contains the calls needed for playing and recording dynamic media. It communicates with the necessary components for the type of media being used.
- *Component Manager*. Previously called the Thing Manager in internal circles, this piece of QuickTime provides a high-level interface that allows applications to communicate during run time with a collection of software objects. These components, affectionately called Things, provide a variety of functions. At present, these functions include image compressing and decompressing, movie data handling, video digitizing, and playback controlling.
- *Image Compression Manager*. This tool handles the interaction among the components that compress and decompress image data. Its services are available both for movie making and playing and for the compression and decompression of still images.

MOVIE TOOLBOX

The basic component of QuickTime is the *movie*. At its highest level, a movie contains one or more *tracks*, each of which points to data of one type (see Figure 1). A movie also includes its time scale, duration, size, location and poster information, current selection and insertion point (if any), preferred volume, image scaling and positioning matrix, and other information (more on this later).

take a Media can take a Track." After sixty-one nights and sixty days DunKennsan left.

From *DunKennsan, the Favorite Disciple*, by Lord James Batson. Any resemblance to any real person or event is intentional and should be construed as such.•

In the future, QuickTime will be able to do things like control audio-visual equipment and manipulate custom-defined types of data (such as scientific instrument data). Also remember that QuickTime will become even more powerful when compression and decompression hardware becomes cheaper and is found in most users' computers.•

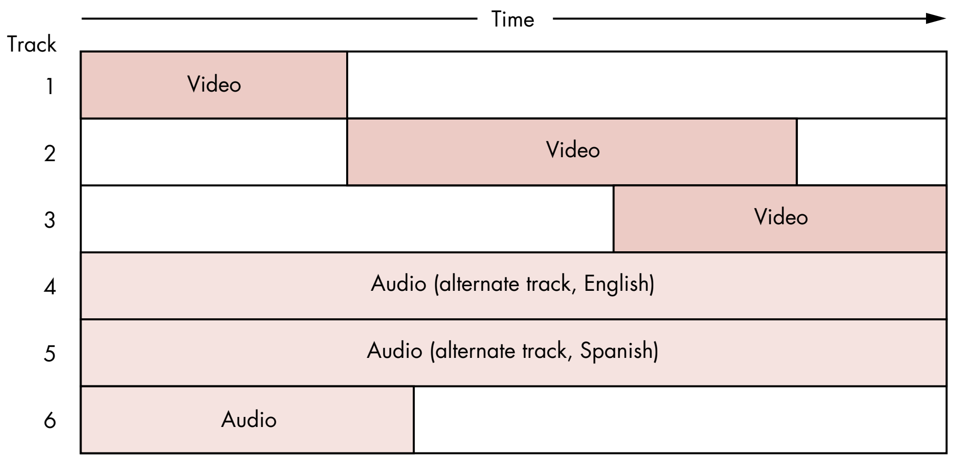

Time
Track
1
2
3
4
5
6
Video
Video
Video
Audio (alternate track, English)
Audio (alternate track, Spanish)
Audio

Figure 1
A Movie and Its Tracks

A movie contains any number of tracks (it's true that a movie can have zero tracks, but that's kind of a boring case). Each track has a *media* associated with it, which points to the "raw" data that the track draws from when it plays. Other track parameters include time scale, duration, time offset within the movie, audio volume, and track type. The *track edit list* is the list of media subsegments that define the track's output.

Each media references a file that contains its raw data; the file can be any place you can put a random-access stream of data—it can be in the file containing the movie, a nearby file, or even a file elsewhere on the network! If more than one media in the movie references the same data file, the different types of data may be interleaved within the file.

As Figure 2 shows, each media is associated with exactly one track and vice versa. Because the track can map nonlinearly to the media (as is the case in Figure 2), you can edit a movie by simply changing a few pointers rather than having to move large pieces of data around. Two or more tracks can be members of a movie's *alternate group*; when the user picks one of these tracks to be active, QuickTime does not use any of the other alternate tracks.

Each media references "raw" data of one type—for QuickTime 1.0, either video or sound. It also contains its duration, time scale, priority, language, quality, media type, and handler. The media handler knows how to play back its data at the right time.

The *poster* is the single frame (in the movie) that the creator of a movie considers as best conveying the spirit of the movie. You can think of it as the frame you would like to show if motion were not possible—for example, when printing the document that

One common use for alternate audio tracks is to let the user watch the movie in the (human) language of her or his choice—for example, tracks 4 and 5 in Figure 1. A movie could also contain alternate video tracks—for example, tracks to be played using hardware decompression (for one track) or software decompression (for the other). In such a case, the video would play on any Macintosh with QuickTime, but it would play better on one with hardware decompression.•

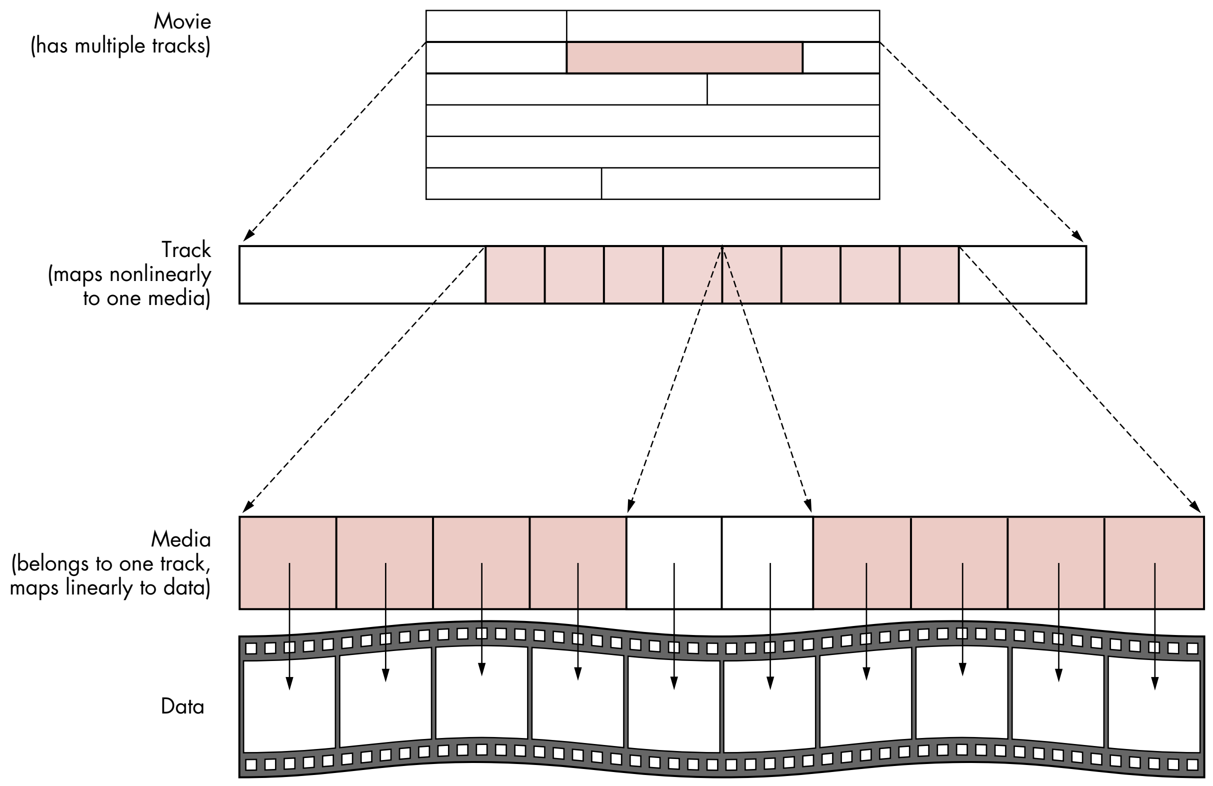


Figure 2
Basic Components of a Movie

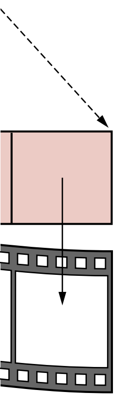
contains the movie. The poster is usually a frame from the movie the user sees, but it can be an arbitrary frame from a video track that is not visible in the normal movie.

The *preview* is a short piece of the movie that best conveys the spirit of the movie. Note that although a preview is associated with a movie, the data (frames) associated with the preview may not be part of the movie the user sees. In movie terms, the track associated with the preview may not be part of the regular movie playback.

Although in the normal situation a movie file contains the data for its tracks (in which case it's called *self-referenced data*), it's possible for the data associated with a media to reside in a file separate from the movie, anywhere on the network.

Future releases of QuickTime are expected to extend the referencing capabilities of media to allow for data being acquired as the movie plays along—as, for example, data coming from a CD player or a video digitizer board.

To recap: A movie may contain any number of tracks. These tracks do not need to be playing at the same time, and as a matter of fact, a track doesn't need to become active at all. Several tracks can belong to a movie's alternate group, and only one of them can play at a time.



COMPONENT MANAGER

One very important architectural feature of QuickTime 1.0 is its extensibility. Let's take a video track as an example. When the Movie Toolbox (the subset of QuickTime that deals with movies) finds out it needs to play back this track, it calls the video media handler (which is a component). The handler in turn calls the Image Compression Manager, telling it the type of compression used. The Image Compression Manager then calls the Component Manager to find out if a corresponding decompressor component is available. If so, the Image Compression Manager can use this component without having to know all the details about the particular decompressor component needed. Of course, this is just one example; several different compression and decompression techniques are available, and the Component Manager allows the caller to choose a certain type of component by supplying additional information about it.

Let's study the decompressor component with subtype 'rpza', which has the following structure:

```
ComponentResource:
ComponentDescription      /* Registration Parameters */
   componentType:              imdc
   componentSubType:           rpza
   componentManufacturer:      appl
   componentFlags:             0x00000447   /* binary 0100 0100 0111 */
   componentFlagsMask:         0
resourceSpec              /* resource where component code is found */
   type:  cdec            /* the code is in a resource of type 'cdec' */
   id:    0x000A          /* with id of 10 */
resourceSpec              /* resource with name string */
   type:  STR             /* 'STR ' resource */
   id:    0x000B          /* with id of 11 */
resourceSpec              /* resource with info string */
   type:  STR             /* 'STR ' resource */
   id:    0x000B          /* with id of 11 */
resourceSpec              /* resource with icon */
   type:  ICON            /* 'ICON' resource */
   id:    0x000B          /* with id of 11 */
```

The registration parameters allow the Component Manager searching for a component of type 'imdc' (image decompression) to narrow the search to a component of subtype 'rpza', made by 'appl' (Apple Computer, Inc.). The parameters include the componentFlags and componentFlagsMask fields, which help determine how to search for a given component. Note that the subtype field can be omitted if no more information is considered necessary for the type of component in question.

For example, the componentFlags field in the example above indicates that the decompressor can do the following:

bit 0	scale on decompress
bit 1	mask on decompress
bit 2	use matrix for blending on decompress
bit 6	spool (used for compression and decompression)
bit 10	do fast dithering

The cleared bits have meaning, too. For example, the cleared bit 3 means that this component cannot use a matrix for the placement and scaling of the decompressed image.

The ComponentResource (shown above) also contains the type and ID of the resource where the code that performs the actual work is located. In addition, it contains the type and ID for resources containing the name string, info string, and icon associated with the component.

In short, the Component Manager can help applications access certain services by function rather than by name; Figure 3 shows how an application can call the Component Manager to interact with different types of components. When an application registers a component, it's guaranteeing that the component supports the basic set of calls defined for the type. This enables applications to find components by their function without having to know exact names or locations.

COMPRESSION AND DECOMPRESSION

Following the basic concepts of the Component Manager, the Image Compression Manager provides applications with a common interface to compression and decompression "engines" that's independent of devices and drivers. Figure 4 shows how the Image Compression Manager interacts with the Movie Toolbox, the Component Manager, and the application.

The services provided through the Image Compression Manager allow applications to compress still images as well as sequences of images (such as those found in video track media). In the case of image sequences, the Image Compression Manager also provides optional support for the *differencing* of frames—that is, storing only the pixels that differ from the previous frame to reduce the size of the movie data.

Given that these compression techniques are tightly coupled to the type of data they're supposed to handle, the Image Compression Manager does not work for sound, text, or any type of data other than images.

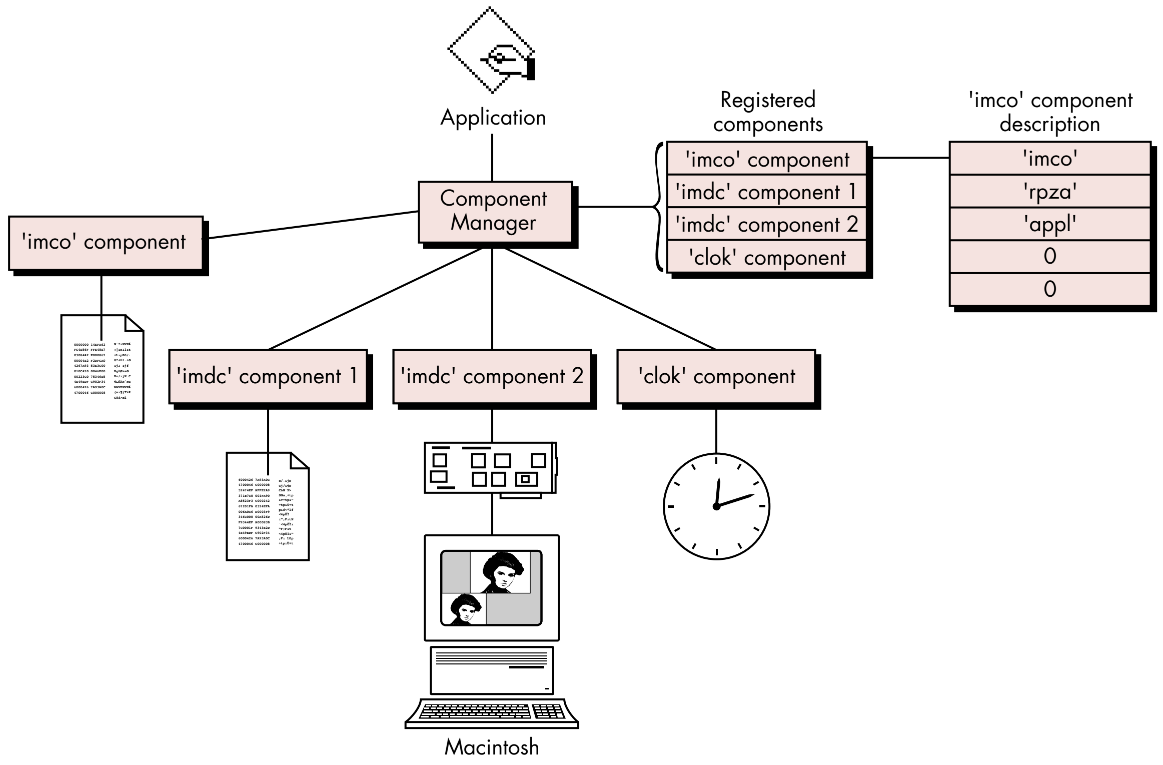


Figure 3
Component Manager Interactions

The Image Compression Manager accepts the input data as either a PICT or a pixMap; obviously, the first format is most often used for still images and the second for sequences of video. Since images can be very large (even when compressed), in both cases the Image Compression Manager allows for the calling application to provide spooling routines that feed the Image Compression Manager source data as needed and write the resulting compressed data to disk. The Image Compression Manager can also translate between pixMaps of varying bit depth. This simplifies the manipulation of an image split across monitors of two different bit depths; it also extends a compressor or decompressor's ability to manipulate images that (because of incompatible pixel bit depths) it would otherwise not be able to handle.

In the case of pictures, the Image Compression Manager provides a set of high-level calls that allow applications to compress and play back PICT resources and files. Although these compression facilities are available to applications that call them, even applications that know nothing about QuickTime's compression facilities can play back pictures containing compressed images. (This can occur because QuickTime-unaware applications calling DrawPicture will automatically invoke the

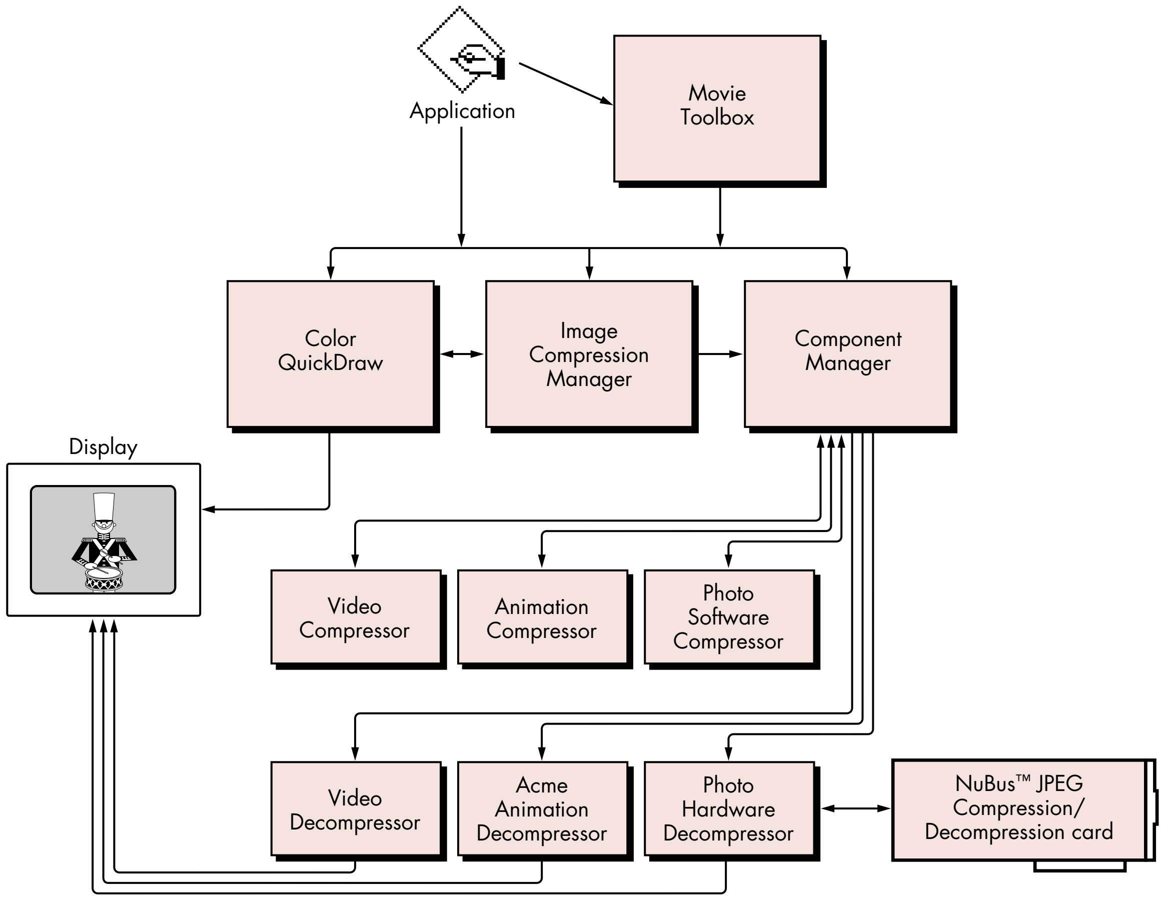

Application
Movie Toolbox
Color QuickDraw
Image Compression Manager
Component Manager
Display
Video Compressor
Animation Compressor
Photo Software Compressor
Video Decompressor
Acme Animation Decompressor
Photo Hardware Decompressor
NuBus™ JPEG Compression/ Decompression card

Figure 4
Image Compression Manager Interactions

Image Compression Manager, which will decompress the image automatically and hand the application the uncompressed PICT image it was expecting.) In other words, when QuickTime is present, you can use compressed PICTs as part of your application and know that any PICT-reading application can open them correctly.

The Image Compression Manager provides a simple and at the same time powerful system for compressing images. Since the mechanism is based in the workings of the Component Manager, adding new compression engines is as simple as dropping a 'thng' file into the Extensions folder of the System Folder (for System 7.0, or into the System Folder itself for System 6.0.7). Even when the exact decompressor component is not available to decompress the data, the Image Compression Manager will find a substitute if any is available. High-level calls are provided for applications to access these features in a nearly effortless manner.

QUICKTIME SAMPLE CODE

We'll now directly explore the QuickTime features that you can immediately put into your applications. We'll follow two samples, each of which accomplishes one of the two basic QuickTime functions: *playing back* a movie (which most applications should be able to do) and *creating* a movie (which you'll need to know how to do if your application creates new movies).

PLAYING BACK A MOVIE

To show the basic steps necessary to open movie files and play them back, we'll use the sample application SimpleInMovie. (You can find the source code for this on the latest *Developer Essentials* disc.) This program presents the user with a dialog for opening a movie and plays the movie back in a window. SimpleInMovie uses QuickTime's standard movie controller (which is itself a component) to let the user start or stop the movie as well as scan back and forth within it. Some commands for the movie controller are implemented as menu commands to show how a program can control the controller component.

But first, a few words . . . Before we look at the SimpleInMovie source code, we need to make several new distinctions. The most important distinction is that of a *public movie* versus a *playable movie*. A playable movie is what the Movie Toolbox manipulates; it has all the information needed for it to be played or edited. In contrast, a public movie is used only for data interchange, and it contains all the information needed to create a playable movie. A playable movie must be converted to a public movie (which is stored as a resource of type 'moov', pronounced "moo-vee") before it can be stored to disk or put into the Clipboard. QuickTime provides two calls to convert between the two forms: GetMoviePublicMovie converts a public movie into a playable movie, and MakePublicMovie does the opposite.

To summarize, a playable movie is what the Movie Toolbox plays back; it has all the media handlers instantiated and is ready to go. A public movie is strictly a static representation used when the movie is to be transferred or copied.

QuickTime gives you wide latitude in choosing the location of the raw data associated with a media, so we need to look at a few alternatives. A movie file has a file type of 'MooV'. We'll call a movie file "normal" if it contains exactly one 'moov' resource.

A movie file whose data fork contains only the media data referenced by the movie and no more is called a *flattened movie*. Specifically, it does *not* contain media frames that aren't referenced by the track to which they belong—for example, the unshaded media frames in Figure 2. A flattened movie is handy for transporting a movie *in toto* to another Macintosh computer. QuickTime provides a FlattenMovie call to create such a movie file.

A movie file containing multiple 'moov' resources would be atypical. In such a case, the application must inform the user of the existence of multiple movies and give the user a way to choose one of them. The Movie Toolbox can handle such files, but Apple discourages you from creating them. Most applications will just use the first 'moov' resource found in the movie file.•

The *single-fork file* is another type of movie file. Here, not only the media data but also the 'moov' resource data are in the file's data fork. (You might use a single-fork file when exporting to a non-Macintosh computer that doesn't have separate data and resource forks.) You can make a single-fork file by calling FlattenMovie with the proper parameters. QuickTime can automatically read these files.

Another possibility is that the movie's media point to data that are not in the movie file's data fork but in a different file; this is very common when you're about to edit a movie. Remember that to edit a track, you need only change pointers to the media; if you had to cut/copy/paste the actual image data (which can be multiple megabytes in length), editing operations could take an inordinate amount of time and disk space.

Back to the code. We can now proceed to examine SimpleInMovie's source code. Note that in the listing below, the comments help describe only those calls that have directly to do with playing movies. The full source code of this program (on the *Developer Essentials* disc) contains numerous other comments on the details that pertain to all normal Macintosh operations.

As is the case with most parts of the Macintosh Toolbox, the Movie Toolbox has to be initialized. In our sample, the initialization is done as follows:

```
void InitMovieStuff()
{
ComponentDescription  controllerDescriptor;
long                  version;
extern Boolean        DoneFlag;
extern Component      movieControllerComponent;

   /* We have to fill in the fields for the player descriptor in order
      to get the standard movie controller component. */
   controllerDescriptor.componentType = 'play';
   controllerDescriptor.componentSubType = 0;
   controllerDescriptor.componentManufacturer = 0;
   controllerDescriptor.componentFlags = 0;
   controllerDescriptor.componentFlagsMask = 0;

   /* We'll use gMoviesInited as a flag for everything; false means that
      the Movie Toolbox or standard player couldn't be initialized. */
   gMoviesInited = false;            /* so pessimistic */

   if (!(Gestalt(gestaltQuickTime, &version)))
      if (!(EnterMovies()))
         if (movieControllerComponent = FindNextComponent((Component)0,
               &controllerDescriptor))   /* No error means we're OK. */
            gMoviesInited = true;       /* Good! */
```

```
    if (!gMoviesInited) {
        Alert(rBadMooviesALRT, nil);   /* Inform user we're bailing out. */
        DoneFlag = true;
    }
}
```

EnterMovies initializes the Movie Toolbox. In an application, this must be balanced by ExitMovies (or Bad Things will happen to your application). If you're calling EnterMovies from a nonapplication environment (such as an XCMD), *you* must call ExitMovies to balance the calls and ensure that all memory allocated and all globals are disposed of.

Normally, when an application presents a movie, it also wants to give the user some basic control over the playing of the movie. QuickTime provides a tool that lets developers add such control easily: a component called the *standard movie controller* (the horizontal bar at the bottom of the window in Figure 5).

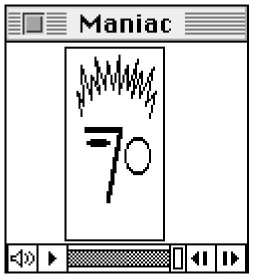


Figure 5
The Standard Controller

Getting and using the component. To use a component, you first have to get it, which means you must fill in a ComponentDescription. Our example specifies only the basic type, but the subtype, manufacturer, and flags fields allow you to specify the component in greater detail. If, for example, you were looking for a compressor, the type would be 'imco' for an "image compressor" or 'imdc' for an "image decompressor." In addition, the subtype could be 'rpza', 'rle ', or 'jpeg' (or others), each of which specifies a specific implementation of compression or decompression.

Although an application can register components "live" (that is, after the application has started up), the normal way they get registered is during system startup, at which time the Component Manager registers all components found in files of type 'thng' in the Extensions folder of the System Folder (for System 7.0, or in the System Folder itself for System 6.0.7). Because this happens automatically, the application can find a specific component by making the following call:

```
FindNextComponent((Component)0, &controllerDescriptor);
```

This call tells the Component Manager to find the component that matches the descriptor (controllerDescriptor); passing 0 in the first parameter tells the Component Manager to return the first one of this type that it finds. In the case of a more extensive search, you may want to continue the search; you would then pass the last component found to get the next in the list that matches the descriptor.

Once you know that the component exists, you have to open it. A component (if it's so designed) can be accessed multiple times simultaneously. Each time a component is opened, the calling application receives what is known as an *instance* of the component. The instance is what the application uses to maintain communication with the component. In our sample, we get an instance of the standard movie controller by calling OpenComponent(movieControllerComponent), where movieControllerComponent is the value returned by FindNextComponent. So keep in mind that there's a difference between a component and an instance of the component.

Once we've done the initialization, the user can select a file, and we can then proceed to set up showing the movie. The code that gets the movie looks like this:

```
if (OpenMovieFile(&(reply.sfFile), &movieResFile, fsRdPerm, nil)) {
   DoReportFailure();
   return;              /* and go back */
}
else {
   if (!(err = NewMovieFromFile(&moov, movieResFile, &resID, nil,
         0, &wasChanged))) {
      if (err = GetMoviesError())
         DebugStr("\perror after NewMovieFromFile");
   }
   else {
      DebugStr("\pCould not get the moov ");
      err = -1;       /* err set will make it skip the rest */
   }
   CloseMovieFile(movieResFile);
}
```

Given an FSSpec (which, in this example, is reply.sfFile), the call OpenMovieFile returns the reference number for the resource fork of the file, once it has been opened. (In the code above, the reference number is in the parameter movieResFile.) It can also return the data reference for the movie, but in this example we pass nil, which indicates that we don't need it; we would need it if we were going to add tracks to the movie. (Later in this article, the section "Creating a Movie" gives more details on data references.)

Once the resource fork is open, we call NewMovieFromFile, which when successful returns the *playable* movie. NewMovieFromFile first gets the 'moov' resource (which is a public movie), creates a movie, and then resolves its data references.

Apple has provided calls such as OpenMovieFile and NewMovieFromFile to simplify things for you. Though it is possible for you to make the low-level calls needed to make a playable movie from a public one, we don't recommend it. You run the risk of confusing the Movie Toolbox, which may result in incorrect values for the self-referenced data references (which indicate that the data is in the same file as the 'moov' resource). Both OpenMovieFile and NewMovieFromFile handle this situation correctly when they resolve the data references.

In our sample, when calling NewMovieFromFile, we pass 0 for the ID, meaning that we'll take the first 'moov' resource found. We also pass nil for a name pointer, since we don't plan to display the name or change it. We proceed to close the file by calling CloseMovieFile. (If we were editing the movie, we would not close the movie file here.)

Now that we have a movie, the next step is to adjust the movie box so that the movie appears in the right place in our GWorld (the window in which the movie appears):

```
GetMovieBox(moov, &moovBox);    /* Get the movie box. */
OffsetRect(&moovBox, -moovBox.left, -moovBox.top);    /* topleft=0 */
SetMovieBox(moov, &moovBox);
```

What is a movie box? Figure 6 shows how the Movie Toolbox calculates the rectangle known as the *movie box* and displays a multitrack movie in an application's window.

- *Pieces 1 and 2*: For each track, the Movie Toolbox takes the intersection of the source rect of the track and the track's clip region (both of these entities share the same coordinate system). The resulting area is transformed into the movie's coordinate system using the track's matrix. In piece 1, the track's clip region is smaller than the image. The clip region of piece 2 is the same size as the track's source rect.
- *Piece 3*. The MovieSrcBoundsRgn is the *union* of all the clipped track regions. In this example, there are two regions. Note that the MovieSrcBoundsRgn includes both the striped and unstriped parts of piece 3.
- *Piece 4.* The MovieSrcClipRgn is the region in which the Movie Toolbox is to display the movie. It clips the image to the areas marked with diagonal and vertical stripes.

- *Piece 5, 6, and 7*. Piece 5 is the image resulting from the intersection of pieces 3 and 4. The MovieBox, piece 7, is the minimum rectangle that contains piece 5, mapped into the local coordinates of piece 6, the MovieGWorld (which belongs to the application that's showing the movie).
- *Piece 8*. The MovieDisplayClipRgn is the last clip applied to the movie before it's displayed.
- *Piece 9*. The application displays only the intersection of pieces 7 and 8, which includes the minimal rectangle enclosing the diagonally and vertically striped areas.

When QuickTime plays a movie, it doesn't take the GWorld's clip region into account; the GWorld's clip takes effect at the level of the application running the movie. If your application draws into the window where a movie is playing, you want to be sure that the MovieGWorld's clip region *excludes* the part being drawn by the Movie Toolbox; in Figure 6, this would be the striped regions within piece 8 *and* the smallest rectangle that contains them.

The last clip applied by the Movie Toolbox occurs when it applies the *movie display clip region* (piece 8 in Figure 6). This clipping area is not, in the strict sense of the word, part of the movie; it is only a run-time option and is not saved in the public movie. (This allows your application to apply a final clip of the movie within your application's GWorld.) If, for example, you used a triangular movie display clip region to clip a larger movie image, the movie would appear in its window as shown in Figure 7.

Now that such an important question has been taken care of, we can go back to the sample code. The main idea here is that the movie box probably does not have its top left corner set to (0,0). So if left to chance, the movie may not be visible in the GWorld (CGrafPort) used to display it, since its coordinate system is the GWorld's. The code then translates the resulting movie to the top left corner of our window, thus ensuring that it will be visible. Figure 8 shows how the movie box can also be used to scale the resulting image.

Our sample application then creates a window for the movie and stores with it the player instance (obtained by calling OpenComponent(movieControllerComponent)) and the movie associated with that window.

Adding the controller. Then we call MCNewAttachedController. The objective here is to put together the movie, the player instance, and the window. Although we recommend that you use the standard controller, it's not the only way to control movies; you can do it all "by hand" if you want tighter control—but you must be careful to do it right.

For your information, the "MC" in QuickTime-related names stands for "Movie Controller."•

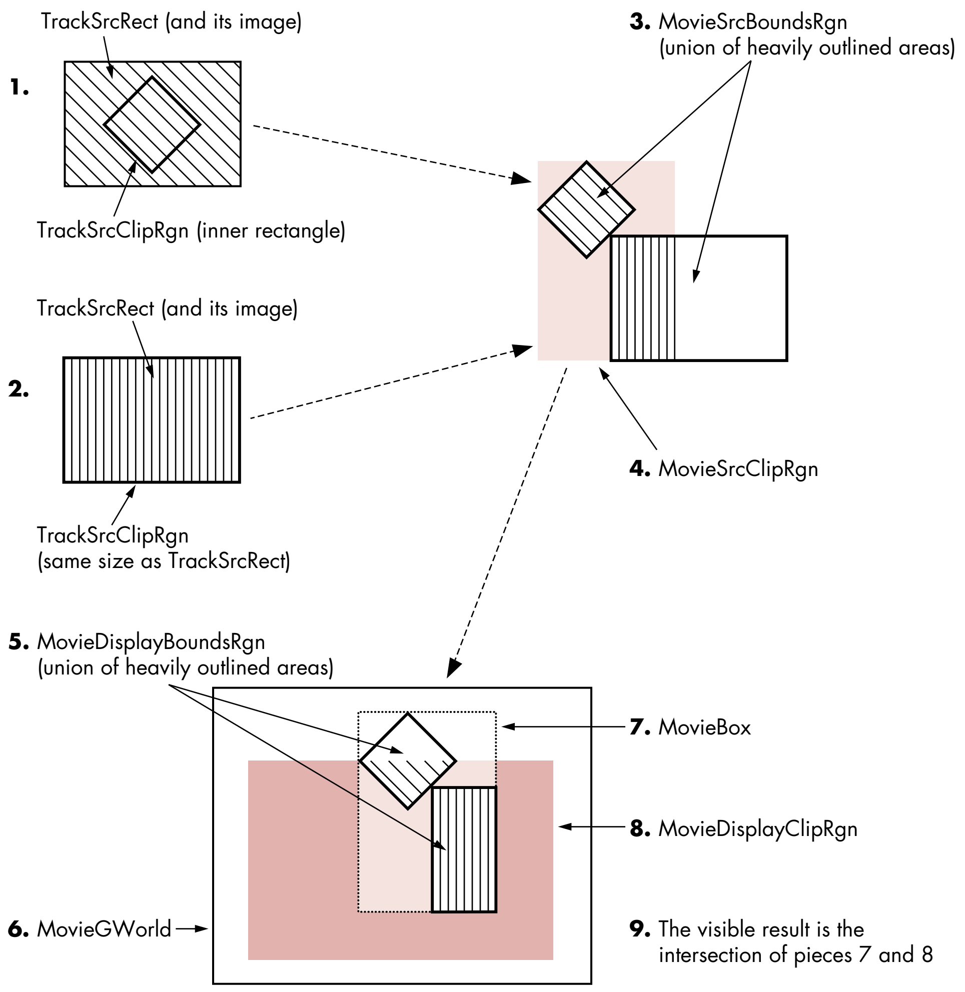

TrackSrcRect (and its image)
1.
TrackSrcClipRgn (inner rectangle)
TrackSrcRect (and its image)
2.
TrackSrcClipRgn
(same size as TrackSrcRect)
3. MovieSrcBoundsRgn
(union of heavily outlined areas)
4. MovieSrcClipRgn
5. MovieDisplayBoundsRgn
(union of heavily outlined areas)
7. MovieBox
8. MovieDisplayClipRgn
6. MovieGWorld
9. The visible result is the
intersection of pieces 7 and 8

Figure 6
Displaying a Multitrack Image in an Application's Window

MCNewAttachedController sets the destination window as the GWorld for the movie and for the drawing of the control, and it attaches the control instance to the movie being played.

Then we must call SetMovieActive to enable the movie to be serviced by calls to MoviesTask; StartMovie then sets it in motion. MoviesTask has to be called periodically (normally as part of the normal idle processing in the event loop) for the movie to display successive frames without erratic playback.

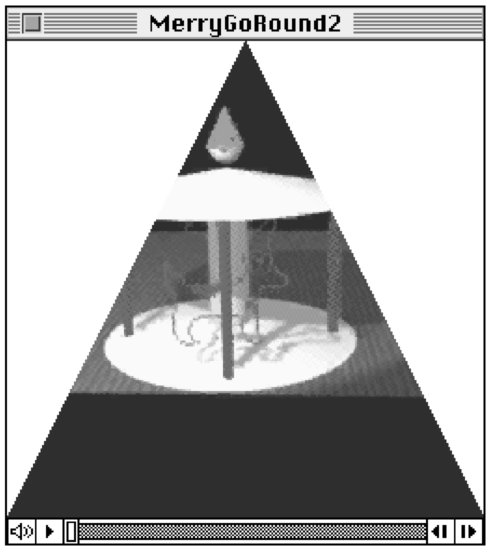
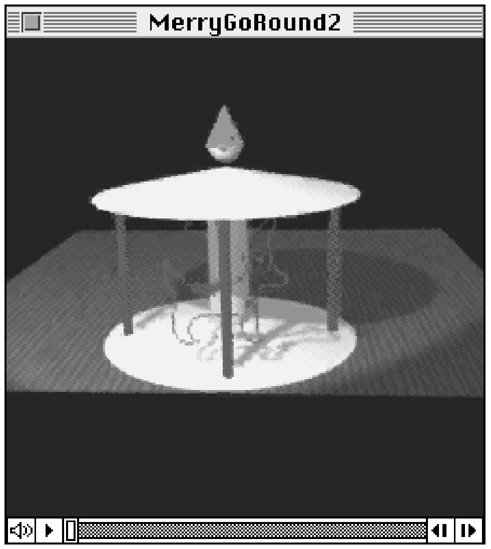
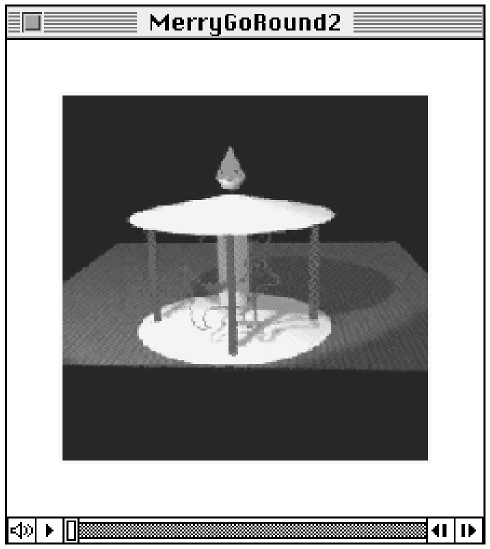


Figure 7
Movie Clipped by Display Clip Region



Original movie



Same movie after
```
InsetRect(&moovBox, 30, 30);
SetMovieBox(moov, &moovBox);
```

Figure 8
Changing the Movie Box

Since in our sample we're using the standard controller, we call instead MCIsPlayerEvent, which accomplishes the following: First it calls MoviesTask to keep the movie (or movies) in motion. Then it performs the tracking of events that belong to the controller itself (such as button clicks and moving the scroll box as the movie moves along).

To summarize, the basics of including playback of movies in an application are as follows:

- Get a public movie and convert it into a playable movie.
- Associate the movie to the GWorld that will display it.
- Set the movie in motion.
- Periodically call MoviesTask to keep the movie in motion.

The above is, necessarily, a simplistic description of the process. By reading QuickTime's documentation, you'll find that the set of calls range from the very high-level, such as when using the standard controller and the movie file calls, to the fine-detail calls that allow you to control the movie at the track level, as well as intermediate-level calls that allow you to control and monitor such movie parameters as the rate (speed) of the movie and its sound level.

CREATING A MOVIE

The sample source code that we'll use to discuss the creation of a movie is called SimpleOutMovies. This program creates a movie that contains two tracks. The first is a video track made out of frames that are read in from PICT files (this is what you must do to create a movie from a sequence put together using a rendering package like MacRenderMan®). The second track uses the data contained in a 'snd ' resource to add sound to the movie.

Again, we won't dwell here on the details that aren't pertinent to the creation of movies. The curious reader is once more invited to check out the source code files on this issue's *Developer Essentials* disc for the whole story. I can't help mentioning that the file-handling part of the code demonstrates how to let the user select a folder and then access the files in it sequentially; I found writing this an interesting exercise.

SimpleOutMovies first calls the usual initialization stuff, including (since we're QuickTime savvy) EnterMovies, the call that initializes the Movie Toolbox. Then it proceeds to prompt the user to indicate where to put the movie file and what to call it, followed by a prompt to find the folder with the PICT files. The program also creates a window to display the frames as they're processed, sizes the window according to the frame of the pictures, and starts the real job.

The code looks like this:

```
theErr = CreateMovieFile(&mySpec, 'TVOD', 0, cmfDeleteCurFile);
if (theErr) DebugStr("\pCreateMovieFile Failed");
theErr = OpenMovieFile(&mySpec, &resRefNum, fsRdWrPerm, &mdrh);
if (theErr) DebugStr("\pOpenMovieFile Failed");
```

After calling CreateMovieFile (passing the FSSpec corresponding to the file the user wants to create), we call OpenMovieFile. Although we discussed this call earlier, there's an important difference: Now we want to use the media data reference (held, in the code above, in the parameter mdrh).

A *media data reference* is what the Movie Toolbox uses to find the location of the data for a given media. In this part of the example, we want to add the data to the same file we're creating. That's why it's important that, when the call returns, the parameter mdrh contains the reference to the file being created. Remember that this type of data reference is called self-referencing because it points to its data as being in the same file as the 'moov' resource. In the case where the data for a track resides in a different file, it's necessary to create a new data reference that points to that file.

Once the file has been opened, we create the movie by calling NewMovie(0, 60). This creates an empty movie (devoid of tracks) but one that you *could* play (with, however, little result). In other words, NewMovie creates what I've been calling a *playable* movie. The parameters indicate that we want the movie to be created inactive and that the TimeScale for the movie is 60—in other words, each unit of time in the movie is 1/60th of a second, equivalent to a Macintosh "tick." (I might add that each track has a TimeScale associated with it, but the Movie Toolbox takes care of synchronizing the individual times.)

Next, we create the video track based on the following two lines of code:

```
gTrack = NewMovieTrack(gMovie, 0, kTimeScale, frameX, frameY);
gMedia = NewTrackMedia(gTrack, VIDEO_TYPE, mdrh, kTimeScale);
```

A track contains bookkeeping information associated with the track's overall data content. For example, to the new movie track we feed in the following: the movie the track is part of (in the above code, gMovie), the time offset (0), the scale for the track (kTimeScale), and the dimensions of the frames as obtained from the PICT frame (frameX and frameY). kTimeScale in this case is set to 10 (which means that the time unit for the track is 1/10th of a second).

Then we create the media associated with the track by calling NewTrackMedia. The parameters establish the type of the media (currently the types defined are VIDEO_TYPE and SOUND_TYPE; new types will be announced as they're defined) and, of course, the time scale for the media. For the last parameter in our

It's important to note that the time offset is given in movie time; since we want the track to start from the beginning of the movie, we pass 0 to NewMovieTrack. Nothing prohibits a track from starting at a time different from 0; if we wanted this track to start two seconds into the movie, we would pass an offset value of 120. •

example we again pass kTimeScale—same as for gTrack—but this is not required; the media can have a different rate than the associated track. The Movie Toolbox provides many calls that allow you to convert between times and rates for those cases when this is necessary.

The next call to the Movie Toolbox is

```
BeginMediaEdits(gMedia);
```

This call is needed here because we're going to add data to the media; in other words, the data comprising the samples will be moved into the media's data file. We'll see that when adding samples by reference (when the data doesn't move), BeginMediaEdits is not necessary.

Capturing the video track. We're now ready to start collecting samples for our video track; enter the Image Compression Manager, stage left. In most cases it's desirable to compress the images to minimize both the size of the resulting file and the amount of data that needs to be moved when playing back the movie.

After we allocate a buffer that can contain the images we want to use, we call

```
GetCompressionSize(&pm, &r, theDepth, theQuality, codecType, codecID,
                   &maxCompressedFrameSize)
```

The purpose of this call is to find out, using the known parameters for the images, an estimate of the worst-case size for the resulting image. (In the same manner, GetCompressionTime can return information concerning the *time* that it would take to compress the image.)

After allocating the buffer for the compressed data, we call

```
CompressSequenceBegin(&seqID, &pm, nil, &r, nil, theDepth, codecType,
                      codecID, theQuality, mQuality, keyFrameRate, ct,
                      codecFlagUpdatePrevious, imageDescriptorH);
```

The parameter seqID points to a variable where the ID of the sequence is stored. This value is needed to continue adding frames to the sequence. We pass nil for both the previous pixMap and rectangle; this indicates that the Image Compression Manager will allocate the GWorld to keep a copy of the image against which the next frame will be compared. If you wanted to allocate it yourself, you would pass it here.

The overall objective of the code that creates this video sequence is, when going from frame to frame, to store as little information as possible for each new frame. Instead of storing a complete image for every frame, we want to add only the difference between a given frame and the *key frame* (the most recent frame that contains the complete image).

The Movie Compressor component has a built-in decision maker that determines when a new key frame is needed. Nevertheless, based on the expected images, the program can set the maximum number of frames that can be added before a new key frame is needed. In our case, we pass a value of 10 for the keyFrameRate, which means that at least every ten frames a key frame of the entire image has to be added to the sequence. If you want to force the creation of a new key frame *every* frame, you can easily do this by calling SetCSequenceQuality and passing 0 for temporalQuality.

The last parameter to mention is the ImageDescription handle. This handle (which the calling program has to preallocate) is filled in by the compressor and contains all the information necessary to reassemble the image. The ImageDescription handle is required by the media to interpret the data. Later, when we add sound to this movie, we'll see how this is handled differently.

Then our code renders one picture in the off-screen GWorld allocated for this purpose and calls

```
CompressSequenceFrame(seqID, &pm, &r, codecFlagUpdatePrevious,
         *compressedFrameBitsH, &compressedFrameSize, &similarity, nil);
```

followed by

```
AddMediaSample(gMedia, compressedFrameBitsH, 0L, compressedFrameSize,
              (TimeValue)1, (SampleDescriptionHandle)imageDescriptorH,
              1L, similarity?sampleNotSync:0, &sampTime);
```

Very similar to CompressSequenceBegin, CompressSequenceFrame adds more frames to a sequence. Note that we have to pass the sequence ID, the ImageDescription handle, and a VAR parameter named "similarity," which tells how close the current frame is to the previous frame (the values range from 0, which means a key frame was added, to 255, meaning that the two frames are identical). The compressor has one flag, codecFlagUpdatePrevious, which tells the Image Compression Manager to copy the current frame to the previous frame's buffer.

This process is repeated for each frame and, when all the PICTs have been processed, we close the sequence and add the media to the movie:

```
CDSequenceEnd(seqID);
EndMediaEdits(gMedia);
InsertTrackMedia(gTrack, 0L, GetMediaDuration(gMedia), 0L,
                 GetMediaDuration(gMedia));
```

The important call here is InsertTrackMedia. This call is the final link in adding samples to a track. When EndMediaEdits executes, the new data samples are already part of the media. However, the track does not know about the additions that have just been made, and the call to InsertTrackMedia takes care of that. There are numerous implications here, but an interesting one is that a segment of the media can be inserted into the track more than once. Since the time scale of the media and track are the same, we can use the value of the media's duration for the track segment's duration, too.

At this point we have completed the creation of a movie and have added a video track to it, so we end with the following:

```
AddMovieResource(gMovie, resRefNum, &resId, (char *)sfr.fName);
CloseMovieFile(resRefNum);
```

That's it—we have a movie that we can play. But we're missing one thing: sound. This is not a big deal, since adding sound is very much like what we've just done. In the paragraphs below, we'll describe what's different.

Adding sound. Before closing the file in our sample program, SimpleOutMovies, we must include the routine that handles adding the sound. The process is the same as it was for video. In this case, the user is prompted for a file containing a sound resource. When selected, the program reads in the 'snd ' resource and, with that data at hand, we proceed to fill in the sound description record.

In the QuickTime 1.0 release, the Movie Toolbox can deal only with sound data made out of sampled sounds; any other data will make no sense. Future releases of QuickTime will most surely have support for other sound formats. This is why most of the fields in the sound descriptor record have to be filled with zeros; but based on the 'snd ' data, we enter the number of channels, the sample size (in bits), and the frequency of the sampled sound.

Then we start again with

```
NewMovieTrack(moov, (TimeValue)0, kTimeScale, 0, 0);
```

Note that for a nonvideo track, the spatial information width and height must be set to 0. Since we're adding sound that has been sampled at a rate of 11 kHz, the constant kTimeScale has been set to this value.

The call to create the new track is followed by

```
NewTrackMedia(gTrack, SOUND_TYPE, mdrh, kTimeScale);
```

Note that we're still using the same media data reference that we used for the video track; this means that we want to continue adding samples to the same file, since that's where the parameter mdrh points.

After this we call BeginMediaEdits to start adding samples, followed by AddMediaSample, EndMediaEdits, and InsertTrackMedia. The difference in this sequence is that AddMediaSample is called only once; since all the sound data is in one place, we add it all at once.

Finally, what happens when we don't want to add the data directly—that is, when it's in a file and we don't want to copy it over to the movie file? In this case we need to add data by reference and first we need to create a media data reference record. Although these calls are System 7.0-specific, QuickTime makes sure they work when running under System 6.0.7.

In this case, our program goes through the same process as before, asking the user to select a file with sound data in it:

```
SFGetFile(dlgPos, "\pSound file:", nil, 1, &typeList, nil, &reply);
```

where "reply" is an old, trusted SFReply. Once the user selects the file, we call

```
FSMakeFSSpec(reply.vRefNum, 0, (unsigned char *)reply.fName, mySpec);
```

passing a pointer to an FSSpec in mySpec.

When we have the FSSpec that describes our sound file, we have to make an alias to it. We do this by calling

```
NewAlias(nil, &mySpec, &SoundFileAlias);
```

From the alias, we create a media data reference by calling

```
mdrh = NewDataRef(SoundFileAlias);
```

We then use the parameter mdrh to create the media (gMedia) and immediately call

```
AddMediaSampleReference(gMedia, 0, fSize, (TimeValue)1,
      (SampleDescriptionHandle)sndDescriptH, nSamples, 0, &sampTime);
```

Note that we pass 0 for the location of the data within the file; fSize is the size of the samples. If the file cannot be found when opening the movie, the user will be prompted to locate the missing file. Since we are not adding the sample data directly, it's not necessary to call BeginMediaEdits and its companion EndMediaEdits.

We did not have to add the sound all at once; it would have been possible to have added samples in smaller chunks if this had been appropriate. One such example would have been to allow the Movie Toolbox to play the sound track by reading in parts of it as needed (instead of all at once). The other side of the coin is that there would be more accesses to the disk and, in instances when the disk media is slow, this could cause a performance degradation. It's recommended that when you create movies, you perform some tests to find the balance that provides the best results.•

So now you know how to create a movie file and then the movie itself; you start out with an empty shell that you must then fill by creating tracks and media. This sample program also shows how to add media, both directly and by reference.

WHAT'S LEFT?

QuickTime comprises over 500 calls, and it was never the intention of this article to detail them all. We hope that after reading this article, you will see great possibilities for QuickTime and will continue collecting information about this new and exciting technology. Who knows, maybe next time "I'll see *you* in the movies!"

QUICKTIME AND THE HUMAN INTERFACE

DISTILLED WISDOM FROM THE QUICKTIME HUMAN INTERFACE GROUP

The discussions we've had of QuickTime-related human interface issues could fill more than a thousand books; below are the main recommendations for a good interface. Most of these guidelines for using movies come from the maxim "put the user in control." Our user-testing has shown that more often than not you really do need to do the following things to keep your users happy.

- Users should be able to look at the screen and figure out which images are movies.
- A movie should open with its poster showing; if it has no poster, its first frame should be shown. Upon first playing, the movie should make a visual transition from the poster state to the movie state. To get back to the poster, users may reset the movie to the beginning.
- If you allow users to resize movies, the movies should by default maintain their original aspect ratios.
- Where it makes sense, make handling movies as much like handling conventional (static) graphics as possible. For example, in your word processor, resize movies the same way you do pictures.
- Movies should not play when a document is opened.
- Users should be able to find the controls for playing any movie easily.
- It should be reasonably obvious not only how to turn the movie on, but also how to turn it off.
- There must always be an easy and immediate way to stop a movie that's playing.
- There must be at least a sound mute control, and preferably a volume control. The sound tracks of different movies will have different sound levels, and movies will be played back in different environments—some that can tolerate loud playback and others that cannot. Also, it's highly desirable that users have a convenient way to adjust the volume of sounds that accompany the movies. The Sound Control Panel is not judged to be adequately convenient for this purpose.
- In most applications, single-clicking a movie must select it, not play it. This allows users to perform operations on the movie such as Cut, Get Info, "hide controls," "resize," or any number of other operations that your application might support.
- Double-clicking a movie may cause it to play, but only if subsequently single- or double-clicking stops it (you have to test for and ignore any immediate second click because many users double-click reflexively).
- If you don't need single-clicking to select a movie, single-clicking may begin the playing of the movie—just as long as single-clicking also stops it (and you dispose of double clicks, both for starting and stopping).
- Don't mix movies that *play* on single click with movies that *select* on single click.

Thanks to Our Technical Reviewers
Jim Batson, Mark Krueger, Scott "Zz" Zimmerman•

SCORING POINTS WITH TRUETYPE

TrueType, Apple's outline font technology, opens up a world of possibilities for improved handling of text. For example, with outline fonts, users can resize text as they've always been able to resize other objects in drawing programs—by grabbing handles and dragging. This article shows how to program this and other exciting transformations to text.

KONSTANTIN OTHMER AND MIKE REED

The Font Manager in System 7.0 can use TrueType outline fonts, in addition to bitmapped fonts, to produce text on the screen and on a printer. In outline fonts, the appearance of individual characters is defined by outlines, not bitmaps. The TrueType font mechanism is also available as an INIT for System 6.0.7 users.

Your application can take advantage of the special capabilities provided by TrueType fonts to transform text in decorative and useful ways. These transformations include shrinking or stretching text to fit a given bounding box and creating patterned, antialiased text. This article provides routines for accomplishing both of these kinds of transformations. First, though, let's explore in more detail how TrueType fonts differ from their predecessor, bitmapped fonts.

WHY TRUETYPE IS TRULY WONDERFUL

To understand why TrueType is truly wonderful, you first have to understand the trouble with bitmapped fonts. With bitmapped fonts, to generate fonts of sizes for which no bitmap exists, QuickDraw simply picks an available font size according to a gnarly algorithm and stretches or shrinks the bits. Unfortunately, when a bitmap is resized the resulting image is often far from pleasing.

This problem is easily understood: imagine you're shrinking a 1-bit image by a factor of two in the vertical dimension. This means two pixels in the source image combine to form one pixel in the result. If both source pixels are black or both are white, the solution is easy. The problem comes when the two source pixels are different. In this case, since most images on the Macintosh appear on a white background, QuickDraw preserves black. Thus, if either source pixel is black, the result is black.

KONSTANTIN OTHMER AND MIKE REED
Have you seen these two guys? Konstantin and Mike, better known as Jake and Elwood, were last seen driving an old Black & White to the Palace Hotel Ballroom where they performed such hits as "Everybody Needs Some PixMaps to CopyBits" and "Gimme Some TrueType." Frequented hangouts: kitchen after lunch meetings, football field, poker tables, slopes at Tahoe, beach, Fitness Center, center stage. Known contacts: Bruce "Second Hand" Leak, Dave "Know" Good. Distinguishing marks: gym bag, running shoes, soft-edged clip art, smooth text forms. Latest fortune cookie: Ask and you shall receive; this includes trouble. Any information as to the whereabouts of these rascals should be sent to their managers, who are probably looking for them.•

QuickDraw uses this same algorithm for larger shrinks. When an image is shrunk vertically by a factor of eight, if any one of the eight source pixels is black, the resulting pixel is black. Because there is more information in the source than can be represented in the destination, the resulting image often looks ragged and is typically too dark.

A similar problem is encountered when enlarging an image. QuickDraw enlarges images by replicating pixels; thus, the result becomes blocky. There simply is not enough information in the source image to provide a better scaled-up representation. Figure 1 shows a 72-point B in the bitmapped Times® font resized by various amounts.

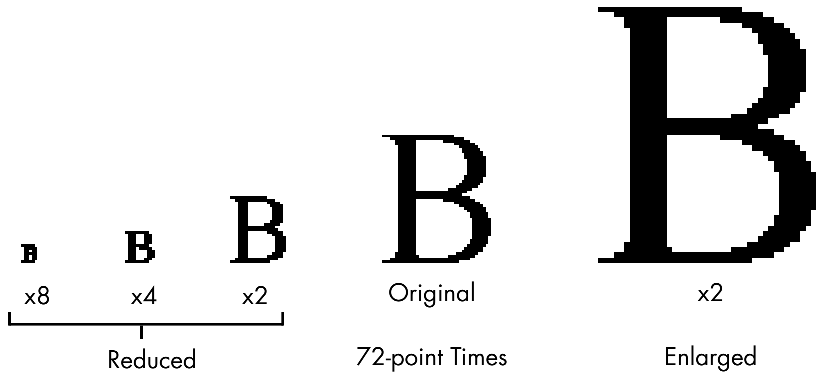


Figure 1
Resizing a Bitmapped Character

This problem with resizing has further implications. Have you ever tried to resize text in the same way you resize other objects in a drawing program like MacDraw®? If you have, you've discovered that when you move the handles on the bounding box, the text clipping and formatting change, but the text isn't resized. This inconsistency between the way text and other objects are handled can be very confusing to users and may deny them the function they really want: to stretch the text to fill the box. The reason for this difficulty is—you guessed it! Although resizing an object such as a rectangle produces excellent results, resizing a bitmapped font results in an image that is, well, ugly.

Enter TrueType. With TrueType fonts, each character is stored as an outline. QuickDraw knows the shape of the character, much as it knows the shape of a circle. When a character is imaged, its outline is scaled to the appropriate size and then filled. When a character is scaled, the outline is scaled rather than the bits. Thus, an excellent representation of the character is possible at all scaling factors. Figure 2 shows how an outline font looks when compared to a bitmapped font drawn in a size for which no bitmap exists.

Note that the fonts shown in the illustrations are drawn at screen resolution, so they don't look as good as they would if drawn on a higher-resolution device such as a printer.•

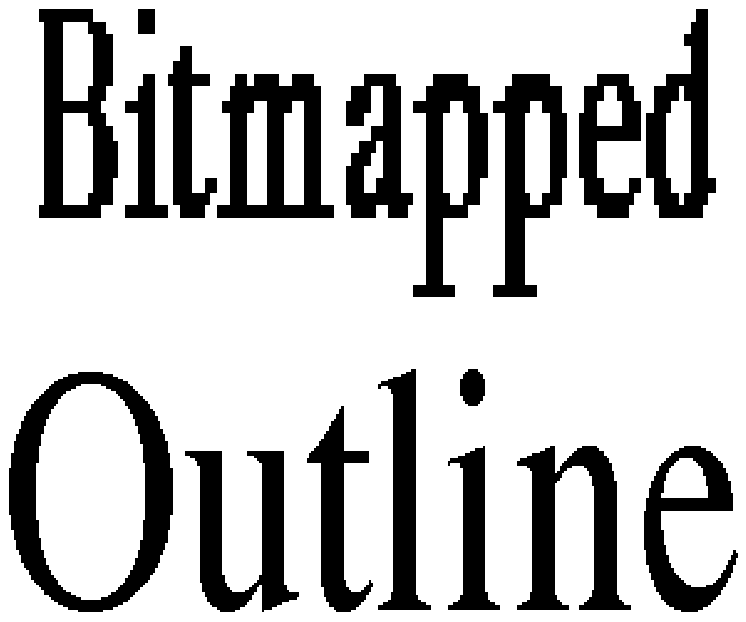


Figure 2
Bitmapped Versus Outline Font

Because characters maintain their integrity at all sizes in TrueType, it's now possible to resize text in the same way users have always been able to resize other objects in drawing programs. We'll show you a routine for doing this. It's also possible to superimpose a text mask on any picture you want in order to create decorative effects. We'll also show you how to do this.

SCALING TEXT TO FIT A GIVEN RECTANGLE

To show how easy it is with TrueType to produce high-quality text scaled to given dimensions, we've written a routine called BoxText that scales a string of text to fit a given rectangle. It takes a parameter that indicates how the text should be constrained: vertically, horizontally, both, or neither. Figure 3 shows examples of text treated in these ways with the BoxText routine.

The BoxText routine is fairly simple. It first checks to see if any constraints are turned on. The constraints are defined as follows:

```
typedef enum boxTextOptions {
   noConstraints,
   constrainH,
   constrainV
} boxTextOptions;
```

The example code shown here was written in THINK C. The *Developer Essentials* disc contains both THINK C and MPW® versions.•

NEW CALLS PROVIDED BY TRUETYPE

In addition to seamless integration with the existing QuickDraw and Font Manager calls (DrawText, GetFontInfo, StdTxMeas, and so on), TrueType offers an extended set of calls. New are three calls that enable applications to choose TrueType fonts over bitmapped fonts, three calls to scale fonts, plus a FlushFonts function to purge the Font Manager's memory caches. Compete explanations of the new calls can be found in *Inside Macintosh* Volume VI; we'll describe the first six briefly here.

```
PROCEDURE SetOutlinePreferred
             (outlinePreferred: Boolean);
```

```
FUNCTION GetOutlinePreferred : Boolean;
```

When a font is selected, the system's default behavior (outlinePreferred = FALSE) is to use an outline font only if there is no bitmapped font that matches the current font, style, and size. If you set outlinePreferred to TRUE, the bitmapped font is chosen only if there is no outline font that matches the current font and style. (Size is not an issue since outline fonts can be scaled to any size.) Note that outlinePreferred is global to the application; the state is *not* kept on a port-by-port basis.

In general, most applications will want to set outlinePreferred to TRUE. The default is FALSE to maintain compatibility with existing applications. Setting outlinePreferred to TRUE allows you to use the OutlineMetrics function (described below) more often.

```
FUNCTION IsOutline (numer, denom: Point) :
   Boolean;
```

The IsOutline function takes scaling factors (like StdText) and returns TRUE if the font specified in the current grafPort will be an outline font when scaled by these factors. One technique for determining whether an outline font exists at all for the current font is to set the text size to 1 point and call IsOutline. This is useful for building a font menu containing only outline fonts.

```
FUNCTION OutlineMetrics (byteCount: Integer;
   textPtr: Ptr; numer, denom: Point;
   VAR yMax: Integer; VAR yMin: Integer;
   awArray: FixedPtr; lsbArray: FixedPtr;
   boundsArray: RectPtr) : OSErr;
```

OutlineMetrics is the successor to GetFontInfo and StdTxMeas. It returns widths, side bearings, and bounding boxes for each character in a given string of characters. Information provided by OutlineMetrics is more detailed and more consistent than that provided by the pre-TrueType font measuring calls.

```
PROCEDURE SetPreserveGlyph (preserveGlyph:
   Boolean);
```

```
FUNCTION GetPreserveGlyph : Boolean;
```

Bitmapped fonts are designed so that all the characters fit vertically between the font's ascent and descent lines. TrueType fonts are not designed with this constraint. To maintain compatibility with the old behavior, the default behavior of the system is to vertically scale down any glyphs that exceed the font's ascent or descent line. If you set preserveGlyph to TRUE, glyphs are not vertically scaled and are drawn at their normal size. (Note that printer drivers set preserveGlyph to TRUE.)

Normally, BoxText works just like DrawString. If you pass constraints, the text is stretched to fit the bounding rectangle's width, or height, or both. This is done through a call to StdText. The stretching factors are computed from the text's original bounds and the bounds passed to BoxText.

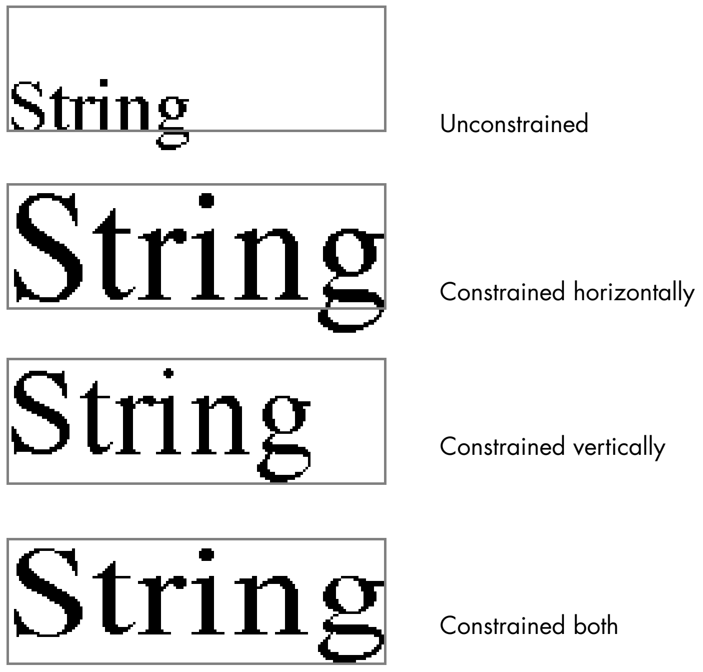


Figure 3
Text Scaled With the BoxText Routine

```
void GetTextRect(char* text, Rect* bounds)
{
   Point    identity;

   SetPt(&identity, 1, 1);      /* No scaling. */
   GetTextBounds(text, identity, identity, bounds);
}

void GetTextBounds(char* text, Point numer, Point denom, Rect* bounds)
{
   FontInfo  info;
   Fixed     hScale = FixDiv(numer.h, denom.h);
   Fixed     vScale = FixDiv(numer.v, denom.v);

   GetFontInfo(&info);
   SetRect( bounds, 0, FixMul(-info.ascent, vScale),
            FixMul(StringWidth(text), hScale),
            FixMul(info.descent + info.leading, vScale) );
   OffsetRect(bounds, thePort->pnLoc.h, thePort->pnLoc.v);
}
```

GetTextRect calls GetTextBounds. GetTextBounds takes a string and scaling factors and returns the bounding rectangle; note that it calls GetFontInfo to determine the height and StringWidth to determine the width. GetFontInfo is used instead of OutlineMetrics since the measurements for the entire font (not just individual characters) are used to calculate the rectangle for the text. Furthermore, GetFontInfo is faster than OutlineMetrics.

The bounds are then scaled by the given scaling factors. The ascent is scaled by the vertical stretching factor to correctly place the text's baseline. Without this adjustment, the top of the text would not align with the top of the constraining rectangle. Finally, the rectangle is offset to the current pen location.

When the text is constrained to fit in the rectangle both horizontally and vertically, the numerator for vertical scaling is set to the height of the rectangle and the denominator is set to the height of the text. For horizontal scaling, the numerator is set to the rectangle width and the denominator is set to the string width. Traditionally, these scaling factors have been stored in point records, and in our routine the code uses SetPt to set the values. Then MoveTo is called to position the pen at the location where the text is to be drawn. Finally, the scaled text is drawn using StdText and the text size is restored.

Note the technique used to call StdText: First the code checks to see whether there are custom bottlenecks in the current port (as there are when printing). If so (the grafProcs field is nonzero), the StdText bottleneck routine, rather than the trap, is called. This is necessary to allow BoxText to print. (Calling the StdText bottleneck is accomplished via the macro, given for both MPW and THINK C, before the BoxText routine.)

```
#ifdef MPW
   typedef pascal void (*StdTextProc)(short count, Ptr text, Point numer,
      Point denom);
   #define STDTEXTPROC(count, text, numer, denom) \
      ((StdTextProc)thePort->grafProcs->textProc) \
      (count, text, numer, denom)
#else    /* THINK C version. */
   #define STDTEXTPROC(count, text, numer, denom)  CallPascal(count, \
      text, numer, denom, thePort->grafProcs->textProc)
#endif

void BoxText(char* myPString, Rect *dst, boxTextOptions options)
{
   Point     numer, denom;
   short     txSize;
   Rect      src;
```

```
if (!(options & (constrainH | constrainV))) {
   /* If there are no constraints, just call DrawString. */
   MoveTo(dst->left, dst->bottom);
   DrawString(myPString);
   return;
}

/* Save the current point size. */
txSize = thePort->txSize;

/* Temporarily set the size to something big, so that our
 * source rectangle is more precise. This is needed since QD
 * doesn't return fixed-point values for ascent, descent, and
 * leading. */
TextSize(100);

MoveTo(0, 0);
GetTextRect(myPString, &src);

switch (options) {

   case constrainH:
      numer.h = numer.v = dst->right - dst->left;
      denom.h = denom.v = src.right - src.left;
      break;

   case constrainV:
      numer.h = numer.v = dst->bottom - dst->top;
      denom.h = denom.v = src.bottom - src.top;
      break;

   case (constrainH | constrainV):
   /* Constrain both dimensions. */
      SetPt(&numer, dst->right - dst->left, dst->bottom - dst->top);
      SetPt(&denom, src.right - src.left, src.bottom - src.top);
      break;
}

if (denom.h && denom.v) {
   /* Since we're applying a fixed scale to src.top, a short,
    * the result, baseline, is also a short. */
   short baseline = FixMul(-src.top, FixDiv(numer.v, denom.v));
   MoveTo(dst->left, dst->top + baseline);
```

```
      /* If there are bottleneck procs installed, call them instead
       * of calling the trap directly. */
      if (thePort->grafProcs)
         STDTEXTPROC(*myPString, myPString+1, numer, denom);
      else
         StdText(*myPString, myPString+1, numer, denom);
   }
   TextSize(txSize);
}
```

RESIZING TEXT INTERACTIVELY

In most drawing programs, you change the size of an object by clicking and dragging with the mouse. This type of interactive resizing is called *rubberbanding* since the borders of the object stretch and shrink like a rubber band. Using the previously described BoxText routine, it's easy to achieve this result for text.

The following routine, SlowRubberBandText, performs the operation. As you can probably guess from the routine name, the performance is not optimal. We'll return to this issue later with the FastRubberBandText routine.

The first thing SlowRubberBandText does is to set the pen mode and text mode to Xor. Xor mode is used so that drawing and erasing can be accomplished without buffering the screen contents and thus without using much memory. The drawback is that the text flickers when it's being resized. A commercial application would check to see if enough memory is available to buffer the screen contents, and if so would provide flicker-free resizing.

Next we have a do-while loop that tracks the mouse as long as the button is held down. On each iteration through the loop, a rectangle is constructed from the anchor point and the current mouse position. This rectangle is drawn and then BoxText is called to draw the text scaled to the rectangle.

The do-while loop waits for the mouse to move or for the button to be let up. If either of these conditions occurs, the text is erased (by being drawn again in the same place). If the button is let up (the terminating condition on the do-while loop), the routine exits, returning the bounding rectangle. Otherwise, the text is drawn scaled, using the new mouse position.

```
void SlowRubberBandText(char* myPString, Point anchorPoint,
                        Rect *theRect, boxTextOptions options)
{
   Point    oldPoint;
   Point    newPoint;
   PenMode(patXor);
   TextMode(srcXor);
   SetRect(theRect, 0, 0, 0, 0);
   do {
      GetMouse(&oldPoint);
      Pt2Rect(oldPoint, anchorPoint, theRect);
      FrameRect(theRect);                      /* Draw it. */
      BoxText(myPString, theRect, options);
      newPoint = oldPoint;
      while (EqualPt(newPoint, oldPoint) && Button())
         GetMouse(&newPoint);
      FrameRect(theRect);                      /* Erase it. */
      BoxText(myPString, theRect, options);
   } while (Button());
}
```

While this routine is a simple illustration of the use of BoxText, it's excruciatingly slow. The reason is that QuickDraw must rerender the outline every time the scaling changes. The FastRubberBandText routine images the text into a 1-bit off-screen GWorld, and then uses CopyBits to stretch the resulting bitmap to fit the specified rectangle. This is similar to the method QuickDraw uses to scale bitmapped fonts, described previously, and is much faster than the slow case.

The code first allocates a sufficiently large bitmap so that the text looks good even at large sizes. If the allocation fails in both temporary memory and the application heap, the code tries smaller rectangles in both heaps. If this also fails, it calls the SlowRubberBandText routine. If the needed memory is available, GWorld gyrations are performed to image the text into the off-screen pixMap. Then a do-while loop similar to the one in the SlowRubberBandText routine is executed, but rather than calling BoxText, it calls CopyBits. The exit conditions are similar to SlowRubberBandText, with the addition that the GWorld is disposed of.

```
void FastRubberBandText(char* myPString, Point p, Rect *theRect, boxTextOptions constraints)
{
   Rect            srcRect, dstRect, origRect;
   GDHandle        oldGD;
   GWorldPtr       oldGW;
   GWorldPtr       myOffGWorld;
   Point           oldPoint;
   Point           newPoint;
   PixMapHandle    myPixMapHandle;
   short           theFont = thePort->txFont;
   short           theFace = thePort->txFace;
   short           err;
   GetTextRect(myPString, &origRect);
   srcRect = origRect;
   OffsetRect(&srcRect, -srcRect.left, -srcRect.top);

/* Scale rectangle up by a factor of 8 to get good results when resizing bitmap. */
   srcRect.right <<= 3;
   srcRect.bottom <<= 3;

/* Take a ride on the GWorld allocation loop. Try temporary memory first,
 * then the application heap. If both fail, keep trying with smaller rectangles
 * until success or until the rectangle is smaller than the original rectangle. */
   do
      if (err = NewGWorld(&myOffGWorld, 1, &srcRect, 0, 0, useTempMem))
         if (err = NewGWorld(&myOffGWorld, 1, &srcRect, 0, 0, 0)) {
            srcRect.right >>= 1;      /* Try rectangle smaller by factor of 2. */
            srcRect.bottom >>= 1;
         }
   while (err && srcRect.right >= (origRect.right - origRect.left));

   if (!err) {
      GetGWorld(&oldGW,&oldGD);

   /* Copy font info from current port into GWorld, clear GWorld, and draw
    * the text into the GWorld. This leaves a pixMap that can be stretched
    * using CopyBits. */
      SetGWorld(myOffGWorld, 0);
      TextFont(theFont);       /* Use font from the current port. */
      TextFace(theFace);       /* Ditto. */
      myPixMapHandle = GetGWorldPixMap(myOffGWorld);
      LockPixels(myPixMapHandle);
      EraseRect(&srcRect);
      BoxText(myPString, &srcRect, constraints);

   /* Back to old GWorld for drawing. */
      SetGWorld(oldGW, oldGD);
      PenMode(patXor);
      TextMode(srcXor);
      do {
         GetMouse(&oldPoint);
         Pt2Rect(oldPoint, p, &dstRect);
         ConstrainRect(&srcRect, &dstRect, theRect, constraints);
         FrameRect(&dstRect);      /* Draw the text scaled to fit in the rectangle. */
         CopyBits(*myPixMapHandle, &thePort->portBits, &srcRect, theRect, srcXor, 0);
         newPoint = oldPoint;
			while (EqualPt(newPoint, oldPoint) && Button())
				GetMouse(&newPoint);
			FrameRect(&dstRect);                /* Erase the text. */
			CopyBits(*myPixMapHandle, &thePort->portBits, &srcRect, theRect, srcXor, 0);
		} while (Button());
		UnlockPixels(myPixMapHandle);
		DisposeGWorld(myOffGWorld);
	}
	else
/* If GWorld allocation fails, use the slow version, which doesn't require a GWorld. */
		SlowRubberBandText(myPString, p, theRect, constraints);
}
```

The FastRubberBandText routine calls ConstrainRect, which mirrors the scaling performed by BoxText. The routine scales the source rectangle to fit inside the destination rectangle with regard to constraints.

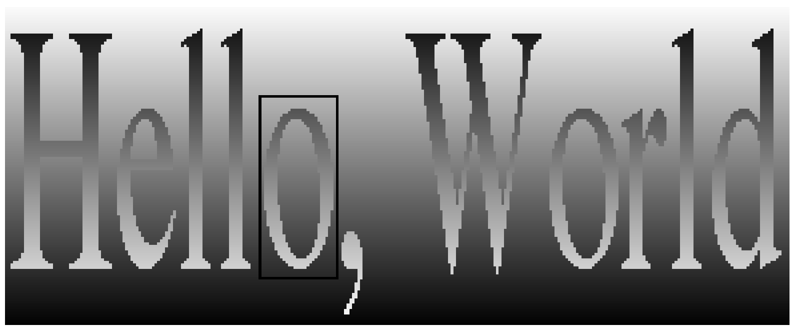
```
void ConstrainRect(Rect* src, Rect* dst, Rect* result, boxTextOptions constraints)
{
   Fixed    ratio;

   *result = *dst;
   switch (constraints) {
      case constrainH:
         ratio = FixDiv(src->bottom - src->top, src->right - src->left);
         result->bottom = dst->top + FixMul(dst->right - dst->left, ratio);
         break;
      case constrainV:
         ratio = FixDiv(src->right - src->left, src->bottom - src->top);
         result->right = dst->left + FixMul(dst->bottom - dst->top, ratio);
         break;
   }
}
```

CREATING PATTERNED, ANTIALIASED TEXT

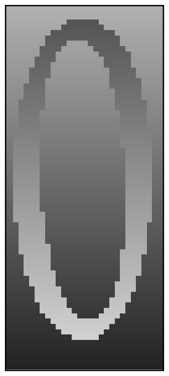
Generation of high-quality scaled text is only one of the fun tricks of the new outline fonts in System 7.0. You can also create patterned, antialiased text with just a few lines of code. (Antialiased text is text whose edges have been smoothed by the addition of gray, creating a softer effect; see Figure 4, and see the antialiased version in color on the inside front cover of this issue.) The possibilities this opens up for writing a "Hello, World" program are staggering, as illustrated in Figure 5.

To achieve this result you use the CopyDeepMask call (available only in System 7.0). Your application generates a source pixMap with the pattern or picture you want to use; the CreateTextMask routine creates a GWorld containing the text mask; and

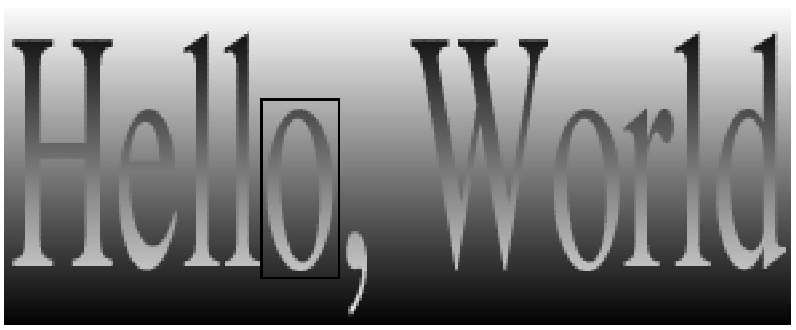
The antialiasing technique used in this article requires a multiple-bits-per-pixel destination device. Since most printers are 1 bit per pixel, these antialiasing techniques are useful primarily for the screen. A second problem with printing antialiased text using these techniques is that QuickDraw does not pass the CopyDeepMask call to printer drivers. For multiple-bits-per-pixel printers, you could image the antialiased text into a GWorld and then use CopyBits to draw the image on a printer.•



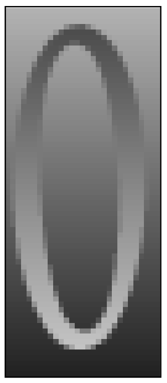
Regular text



Magnified view



Antialiased text



Magnified view

Figure 4
Regular Versus Antialiased Text

finally, you call CopyDeepMask to image the source through the mask onto the destination. Figure 6 illustrates this.

GENERATING THE MASK
The CreateTextMask routine works as follows: First, we attempt to allocate a GWorld that would allow text to be rendered at four times its final size. (If there's not enough memory in the application heap or temporary memory to allocate a GWorld this big, GWorlds of three times and then two times the final size are created. If all of these attempts fail, nil is returned.) Next the 1-bit GWorld is cleared to white and the text is imaged into it scaled by a factor of four (or whatever multiple the 1-bit GWorld turned out to be) in each direction. Then CopyBits with mode

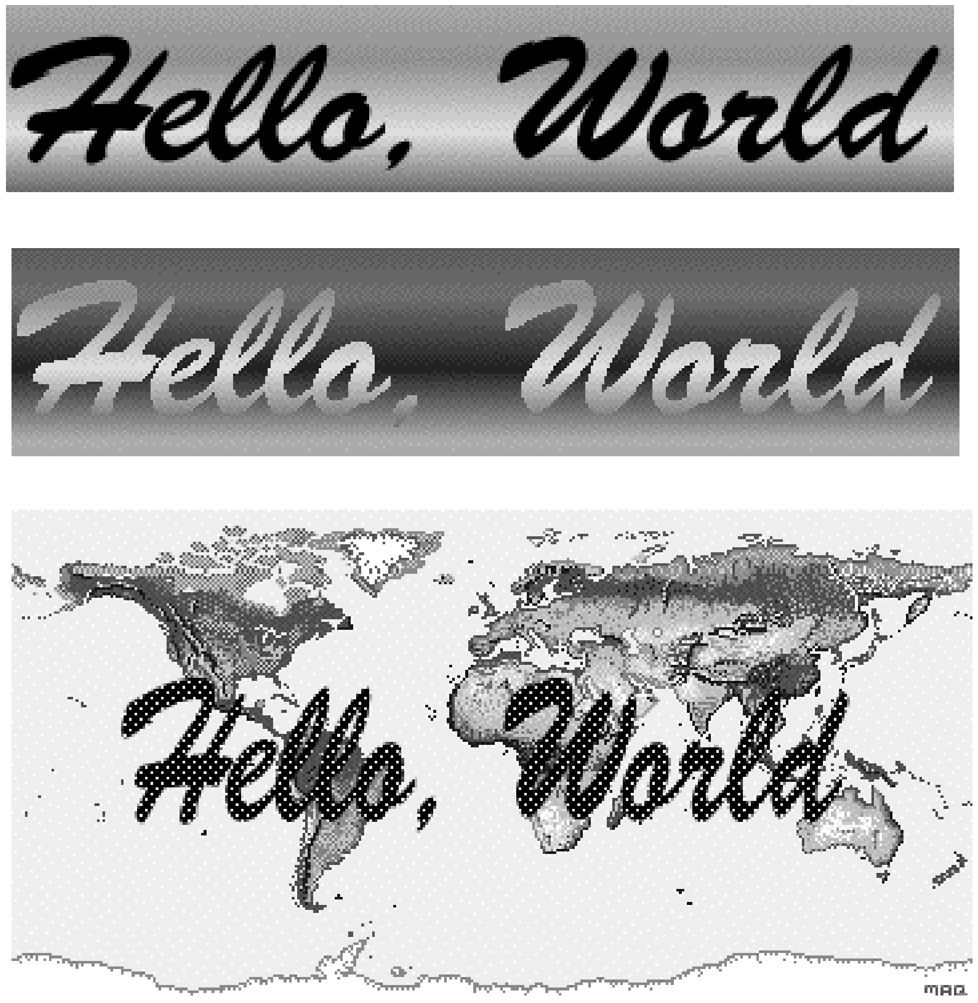


Figure 5
How "Hello, World" Looks in 1991

ditherCopy is used to shrink the large 1-bit GWorld to a 4-bit gray-scale GWorld; this is illustrated in Figure 7.

Because the mask will consist only of grays, the 4-bit GWorld is given a gray-scale CLUT. We are now finished with the 1-bit GWorld and dispose of it. The 4-bit gray-scale image we just created is returned by CreateTextMask. Notice that this routine may return a GWorld allocated in temporary memory, so you must dispose of the GWorld before calling WaitNextEvent.

For details on using CopyBits with ditherCopy, see "QuickDraw's CopyBits Procedure" in *develop*, Issue 6.•

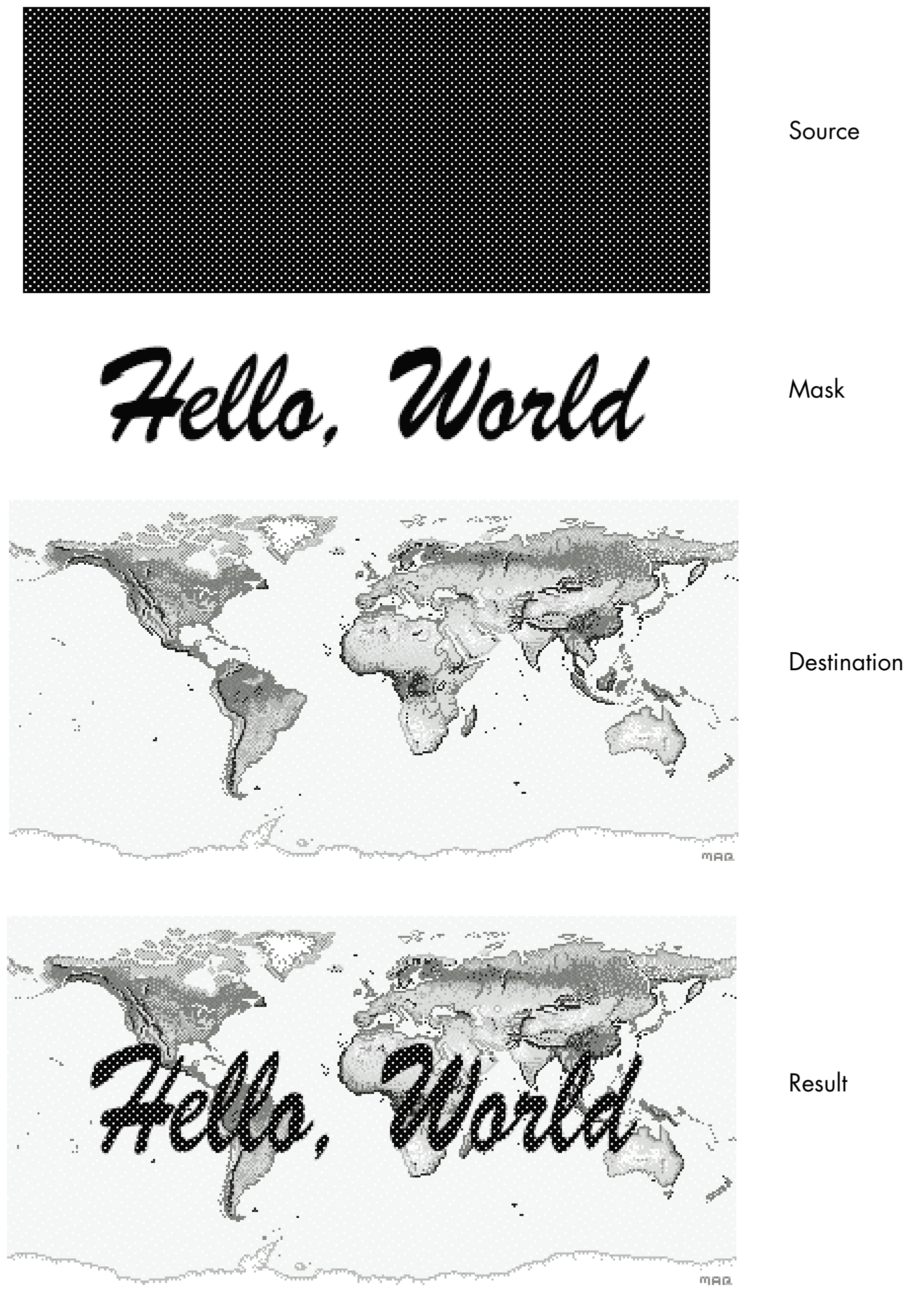


Figure 6
Using CopyDeepMask

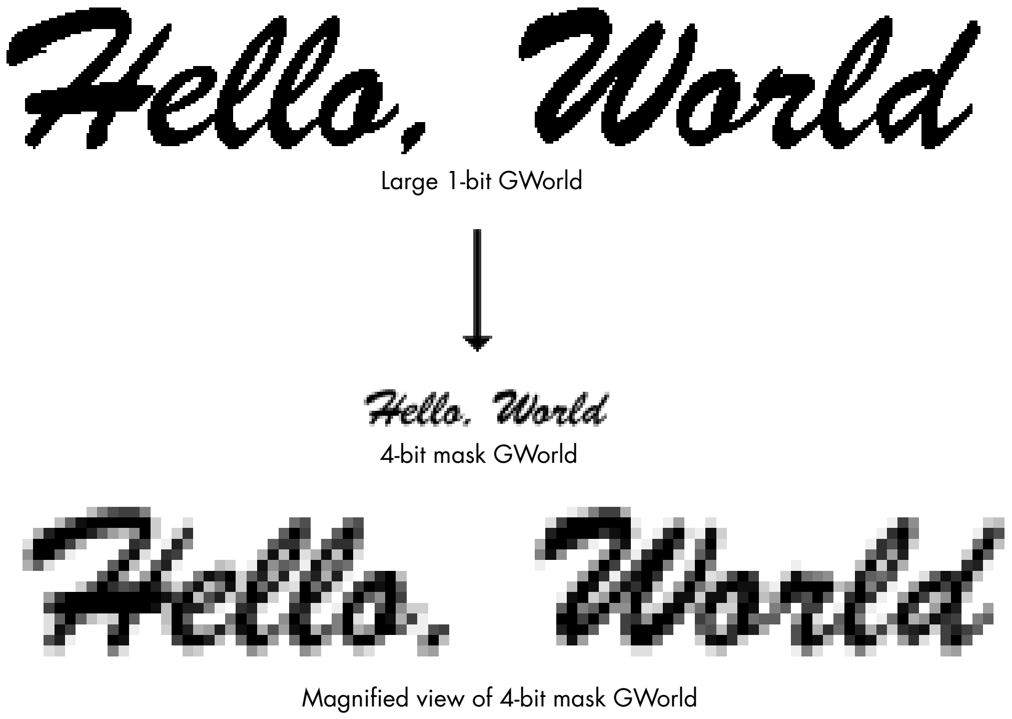


Figure 7
Shrinking the Mask With ditherCopy

```
GWorldPtr CreateTextMask(char* text, Rect* myRect, boxTextOptions options)
{
PixMapHandle    bigPixMap;
GDHandle        oldGD;
GWorldPtr       oldGW, maskWorld, bigWorld;
short           theFont = thePort->txFont;
short           theFace = thePort->txFace;
Rect            myRectBig;

/* Create the 4-bit maskWorld. */
   {  CTabHandle ctab = GetCTable(4+32);
      if (NewGWorld(&maskWorld, 4, myRect, ctab, 0, useTempMem) != noErr)
         if ((NewGWorld(&maskWorld, 4, myRect, ctab, 0, 0) != noErr)) {
            DisposHandle(ctab);
            return 0;
         }
   }
/* Create the supersample bigWorld. First try to use a GWorld 4 times larger. If
 * that fails, try 3 and 2 times larger. If all attempts fail, return a nil GWorld. */
   {  short zoom = 4;
      short width = myRect->right-myRect->left;
      short height = myRect->bottom-myRect->top;
      do {
         SetRect(&myRectBig, 0, 0, width * zoom, height * zoom);
         if (NewGWorld(&bigWorld, 1, &myRectBig, 0, 0, useTempMem) == noErr)
            break;
         else
         if (NewGWorld(&bigWorld, 1, &myRectBig, 0, 0, 0) == noErr)
            break;
         zoom--;
      } while (zoom > 1);
      if (zoom == 1) {
         DisposeGWorld(maskWorld);
         return 0;
      }
   }

/* Draw the text into supersample bigWorld. */
   GetGWorld(&oldGW, &oldGD);
   SetGWorld(bigWorld, 0);
   TextFont(theFont);
   TextFace(theFace);
   LockPixels(bigPixMap = GetGWorldPixMap(bigWorld));
   EraseRect(&myRectBig);
   BoxText(text, &myRectBig, options);

/* Create 4-bit maskWorld by shrinking the big GWorld (with ditherCopy) into the 4-bit
 * gray-scale GWorld. NOTE: This is one of the slowest steps, and is relatively easy to
 * optimize with a custom shrinking procedure. */
   {  PixMapHandle maskPixMap = GetGWorldPixMap(maskWorld);
      LockPixels(maskPixMap);
      SetGWorld(maskWorld, 0);
      CopyBits(*bigPixMap, *maskPixMap, &myRectBig, myRect, ditherCopy+srcCopy, 0L);
      UnlockPixels(maskPixMap);
   }
   DisposeGWorld(bigWorld);
   SetGWorld(oldGW, oldGD);

   return maskWorld;
}
```

SAYING "HELLO, WORLD"

Now we're ready to say "Hello, World" in patterned, antialiased text as shown in Figure 4. The following routine combines the BoxText, FastRubberBandText, and CreateTextMask routines to produce its results. In this example, the source pixMap contains a picture that's read in from a resource file. This pixMap could, of course, contain anything, and that image would peek through the text mask. The interesting thing to notice is that if you scale the source picture to the size of the text, the image behind each letter will stay the same regardless of the scaling factor.

```
void JustShowOff(Point anchorPt)
{
#define      kBoxOptions   constrainH | constrainV

   Rect         myRect;
   GWorldPtr    mask = 0;
   char         *text = "\pHello, World";

   FastRubberBandText(text, anchorPt, &myRect, kBoxOptions);

   if (mask = CreateTextMask(text, &myRect, kBoxOptions)) {

      GWorldPtr       oldGW;
      GWorldPtr       src;
      PicHandle       pic;
      PixMapHandle    maskBits;
      PixMapHandle    srcbits;
      GDHandle        oldGD;

      if (NewGWorld(&src, 8, &myRect, 0, 0, useTempMem) != noErr)
         if (NewGWorld(&src, 8, &myRect, 0, 0, 0) != noErr)
            goto EXIT;
      if (!(pic = GetPicture(1001)))  /* Assumes the PICT is marked purgeable. */
         goto EXIT;
      GetGWorld(&oldGW, &oldGD);
      SetGWorld(src, 0);              /* Set to draw into off-screen 8-bit. */
      DrawPicture(pic, &myRect);      /* Stretch picture to fill user's rect. */
      LockPixels(maskBits = GetGWorldPixMap(mask));
      LockPixels(srcBits = GetGWorldPixMap(src));
      SetGWorld(oldGW, oldGD);
      CopyDeepMask(*srcBits, *maskBits, &thePort->portBits, &myRect, &myRect, &myRect, srcCopy, 0L);
      DisposeGWorld(src);
   }
EXIT:
   if (mask) DisposeGWorld(mask);
}
```

SUMMARY AND PARTING THOUGHTS

TrueType fonts provide high-quality characters at all sizes and scaling factors. This is possible because TrueType fonts are stored as outlines rather than bitmaps, and scaling an outline produces much better results than scaling a bitmap. The BoxText routine described in this article uses the StdText call to stretch text to fit within a specified box. A logical extension of this could solve a problem that has plagued humankind (or at least high school students): how to expand six and a half pages of text to produce a report that's exactly ten pages long, as required by law or a high school teacher.

Usually tactics such as adjusting the margins, line spacing, and font size can get you close. But this trial-and-error process could easily be replaced with a procedure written by some enterprising and humanitarian programmer:

```
FitTextToPages(char *text, long numPages);
```

Rather than simply adjusting the margins and line spacing, this routine could uniformly stretch a block of text to fill the desired number of pages.

System 7.0 also allows applications to pass a deep mask to CopyMask or CopyDeepMask. By imaging text into a large bitmap and then scaling it down with ditherCopy, it's possible to generate a soft-edged mask for producing antialiased text. Furthermore, text drawn using CopyDeepMask can have any image as the source pixMap, making it easy to produce patterned or picture text.

In the process of producing all of these great effects with text, we used GWorlds extensively. From the code samples it should be clear that GWorlds are extremely simple to create and manipulate. Judicious use of temporary memory for holding GWorlds allows our sample "Hello, World" program to run in a 100K heap and still produce very large (bigger than 1200 x 500 pixels) scaled text (provided there's enough temporary memory available, of course).

Enjoy!

Thanks to Our Technical Reviewers
Pete "Luke" Alexander, John Harvey, Bryan "Beaker" Ressler•

Zz speaks

PRINT HINTS FROM LUKE & ZZ

HELP FOR YOUR DIALOG APPENDAGES

OK, so you're cruising your source like a madman trying to get all those little System 7.0 changes in before that target ship date (whatever it is this week), and you notice that the items you've added to the Page Setup/Print dialogs don't have any Balloon Help™—you know, those items like Reverse Pages, Print Hidden Text, or even Use Fractional Font Widths. (Coming up with meaningful names for these items, in four words or less, was a little tricky.) But now there's help, literally, in System 7.0. Great! Then you remember that there wasn't a simple Printing Manager call to add those items to the dialogs. You had to resort to the technique described in Technical Note #95, How to Add Items to the Print Dialogs. As you remember, it used a set of procedures that modified the Printing Manager's (that is, the selected print driver's) dialog item list (DITL) resource.

Remembering the actual code in Tech Note #95, you consider that the Standard File package shipped with 7.0 has a new call that allows you to append things to its dialog, including Help Manager resources. Feeling relieved, you think, "Ah, there must be a similar call in the new Pri . . ." But no, the new printing architecture has been delayed. A quick scan shows that there isn't even a Printing Manager chapter in *Inside Macintosh* Volume VI! Help!!!

It's not as bad as you might expect. If you consult Tech Note #95, you'll see the rather husky AppendDITL procedure. This procedure is called to append the items that you want to add to the dialog item list being used by the particular dialog (Page Setup or Print).

The sample code from Tech Note #95 calls some Printing Manager routines that let you get in after the DITL resource has been loaded, but before the dialog has been displayed. You add your items to the resource in memory *without* calling either ChangedResource or WriteResource. The driver then uses this DITL and displays your items. Once the dialog is dismissed, the resource is purged, and the driver doesn't even know you were there. Life is good.

As you've probably guessed by now, you're going to have to append to the Help Manager dialog item help ('hdlg') resource in the same way that you appended to the DITL resource. You simply scan the list to the end, and then append the appropriate items. The 'hdlg' resource is purged in the same way as the DITL resource, so once again you make no permanent changes.

On the next page is the definition of the Append2hdlg procedure. We start by getting both 'hdlg' resources into memory. It's safe to assume the source 'hdlg' resource hasn't been loaded yet, but we use the SetResLoad trick on the destination in case it has already been loaded. (The SetResLoad trick is a method for determining whether a resource has already been loaded. This trick is important, since in cases where the resource has already been loaded by the system, you don't want to unload it or permanently change any of its resource attributes.) The trick works like this: You SetResLoad to false so that the Resource Manager doesn't load the resource data; instead, it just creates an empty resource handle that can be passed to routines like

SCOTT "ZZ" ZIMMERMAN loves Disneyland—we think it's because he's really a cartoon character at heart. When asked, he admitted to wanting to be Captain Hook when he grew up. His favorite ride is Peter Pan because it's romantic, cool, dark, and the main character is a kid who never grew up. Except for the romantic, cool, and dark parts, it reminds him a lot of life at Apple. When he's at Apple he makes sure he drinks at least 15 gallons of Mountain Dew a day—he says it powers the mechanism for his retractable Barbie Doll hair. In closing, we'll leave you with his favorite question, "How can I miss you if you won't leave?"•

```
void Append2hdlg(srcResID, dstResID)
short srcResID, dstResID;
{
   Handle          srcHdl, dstHdl;
   Ptr             srcPtr, dstPtr;
   short           srcLength, dstLength;
   short           missingItmSz;
   SignedByte      dstHState;

   srcHdl = GetResource('hdlg', srcResID);
   if (srcHdl != nil) {
      SetResLoad(false);                        /* System resource, make sure it's not */
      dstHdl = GetResource('hdlg', dstResID);   /* already loaded. */
      SetResLoad(true);
      if (*dstHdl == 0)
         dstHdl = GetResource('hdlg', dstResID);
      dstHState = HGetState(dstHdl);

      if (dstHdl != nil) {
         srcPtr = (Ptr)*srcHdl + sizeof(hdlgHeader);
         missingItmSz = *((IntPtr)srcPtr);
         srcLength = GetHandleSize(srcHdl) - (sizeof(hdlgHeader) - missingItmSz);

         dstLength = GetHandleSize(dstHdl);
         SetHandleSize(dstHdl, dstLength + srcLength);
         if (MemError() != noErr) {
            DebugStr("\pMemError");   /* Use this error handler, go to jail. */
            ExitToShell();           /* It's the law! */
         }
         dstPtr = (Ptr)*dstHdl + dstLength;
         srcPtr = (Ptr)*srcHdl + sizeof(hdlgHeader) + missingItmSz;

         HLock(srcHdl);
         HLock(dstHdl);

         BlockMove(srcPtr, dstPtr, srcLength);

         HUnlock(srcHdl);
         HSetState(dstHdl, dstHState);

         ((hdlgHeaderPtr)*dstHdl)->hdlgNumItems += ((hdlgHeaderPtr)*srcHdl)->hdlgNumItems;
      }
   }
   ReleaseResource(srcHdl);
}
```

GetResInfo. You then call GetResource on the resource you're looking for. If the handle returned is empty (that is, points to nil), you know the resource isn't already in memory. If the handle returned is not empty, something else (like the system) has already loaded it before you called GetResource. In this example, since we use HGetState and HSetState to preserve the resource attributes, and we want the resource to be left in memory when we're done, we don't really need to know if it was already loaded. The SetResLoad code is included for anyone who is planning on modifying this code to do more.

Next we initialize our locals. We want to point srcPtr to the place in the source resource that we want to start copying from. To do this, we need to point past the "missing item." The size of the item is stored in the resource just after the resource header. We first use srcPtr to get the size (in bytes) of the missing item. Using that size, we can calculate the starting location for the copy. We don't actually initialize srcPtr yet, since resizing the destination handle could move memory. Next we initialize dstPtr and dstLength. In the process, we resize dstHdl to make room for the items we're going to append. Once SetHandleSize has been called, we also initialize srcPtr.

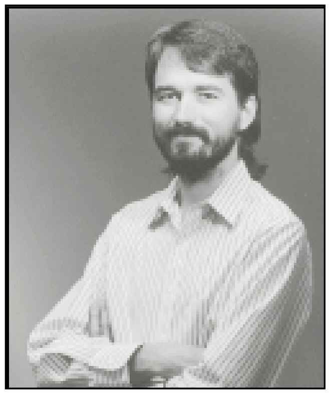
Now that srcPtr and dstPtr are set up, we use BlockMove to copy the new items into the destination resource. After unlocking the resource handles, we update the numItems field of the destination resource so that the Help Manager will know how many items we added. Finally, we release the source resource. We don't want to release the destination, since our changes would then be lost.

So that's about it. Append2hdlg is a lot smaller than AppendDITL because we don't actually need to parse the contents of the 'hdlg' resource. Although it's another piece of code to be added to your application, this should be quite painless, unlike other Printing Manager exercises. Don't forget to read the Help Manager chapter of *Inside Macintosh* Volume VI for guidelines on the contents of your Help Manager balloons. See ya next time . . .

For more information on the format and use of the "missing item" in an 'hdlg' resource, and much more about Balloon Help, see the Help Manager chapter in *Inside Macintosh* Volume VI. •

THREADED COMMUNICATIONS WITH FUTURES

Interprocess communication (IPC) promises to provide a solution to problems that can't ordinarily be solved in a single-tasking, single-machine environment. But attempts at implementing IPC with traditional programming techniques lead to cumbersome code that doesn't come close to realizing IPC's potential. This article shows an example of using threads and futures to do IPC in a way that allows you to achieve concurrency with clean, robust code.



MICHAEL GOUGH

In the article "Threads on the Macintosh" in Issue 6 of *develop*, I identified a potential problem with interprocess communications when you're using the client-server model. Simply put, if you don't use threads when you're using the client-server model to implement IPC, the result could be deadlock. The deadlock occurs because each application is capable of only a single train of thought. The client expects an answer to a question posed to the server but never receives that answer because the server must receive an answer to its question before it can respond. The result is that each party is waiting for answers to its own questions before it can proceed.

Although "Threads on the Macintosh" sounded the alarm about the communications deadlock problem, it didn't go into detail about how threads can be applied to solve the problem. That's the purpose of this article. Specifically, this article shows how you can avoid client-server deadlocks by using threads and a new facility called *futures*. The Futures Package has been integrated seamlessly with Apple events. In this article, we'll use Apple events as the generic facility for implementing IPC. The sample code presented here appears on the *Developer Essentials* disc for this issue.

Before discussing futures in detail, let's review some of the basics about threads. Threads provide multiple trains of thought for your application. If your application is doing more than one thing at a time, threads allow you to simplify your code. Instead of juggling between multiple tasks, you start a separate thread to handle each individual task. You then have multiple program counters, one for each thread. Of

MICHAEL GOUGH has been ranting about threads, futures, and other stuff for about three years at Apple. (Maybe he'll pipe down if we let him publish one more article.) Here are a few things you might not know about Michael: While he was at NASA, he developed solid model generation and ray tracing software. He also introduced virtual memory and remote procedure call support for NASA's Massively Parallel Processor. Michael developed a package called "Virtual Data Table" which makes complex multidimensional data easy to deal with. His pride and joy is a package called "Spherical Database" which turns geographic data sets into a continuous function over a sphere. Michael is often seen levitating things around the office and doing other magic tricks. He insists he learned the levitation trick while traveling in India, but we

course, the threads don't actually run simultaneously on a single CPU. They share the CPU, cooperatively trading control by calling a special function that says, "Let the other threads in this application have some CPU time."

HOW THREADS AND FUTURES FACILITATE IPC

Ideally, when you're writing code for IPC, you'd like things to work such that whenever the client poses a question, it gets an immediate answer. This situation would translate into nice linear code, such as the following:

```
•
•  code that prepares the question
•
answer := Ask(question);
•
•  code that uses the answer
•
```

The semantics would be very simple: A question is prepared, and then it's "asked." The Ask function waits synchronously for the answer to be returned. When it's returned, execution continues and the answer is used.

Unfortunately, this code suffers from a fatal flaw: the synchronous nature of the Ask function will cause a deadly embrace in some situations. What if, as we saw above, the client never receives an answer because the server needs to ask something of the client before it can reply? This is an all too common situation.

Threads allow you to circumvent the deadlock problem by making each application have a client and server portion so that both sides can ask and answer questions of each other. In other words, the client and server portions of each application are assigned to separately executing threads. IPC then works as follows: Application 1's client asks a question of application 2's server, and application 2's server must ask a question before it can answer. However, application 1 is able to field this question because even though its client portion is waiting for an answer, its server portion is available to answer application 2's question. Because the answering and the questioning portions of each program are able to function independently, a hangup in the client or the server doesn't bring the application to a halt.

In the "plain threads" situation just described, notice that execution of the client thread is still held up while the client is waiting for an answer. What futures do is to postpone or even eliminate this delay in processing, giving the thread a chance to do related work before blocking. In this way, futures extend the capacity of threads to maximize the efficient use of the CPU.

think that's balderdash. When Michael's not at work, he's hiking and eating ice cream with his two sons. On occasion, he visits Nevada to take the casinos for all they're worth. When he's done with his current project, he's going on a really long trip to Antarctica.•

In the futures implementation, when a question is posed, the application never has to wait for an answer; it can continue execution immediately. This may seem impossible: in the above example, how can the Ask function return immediately when it must supply an answer to the question? Mustn't it wait until the answer is received? No, because the answer that's returned by Ask is a future. The future doesn't contain the information that the real answer contains. Instead, it contains information that says "this answer isn't 'real' yet." Your code keeps executing, thinking that it has the answer, but it really doesn't. At some point later, when the real answer is received by the Apple Event Manager, the future is automatically transformed into the real answer, with all the information that was requested.

Note that when it comes time to get the contents out of an answer, and the answer could be a future, you must be executing in a thread other than the main event loop thread, or the result will be deadlock. This is because a thread that attempts to access the contents of a future is blocked until the real answer is received. And since the real answer to a future is received by the main event loop, you can't risk blocking the main loop by using it to access the future. The solution is to fork a thread to access the future. This way, your main event loop keeps running, receiving Apple events and passing them to the Apple Event Manager.

In some situations you'll need to find out whether an Apple event is a future or not, and you'll need to do this without blocking. This is done through a call to the IsFuture function. It returns a Boolean value of TRUE if the Apple event you passed

THE POWER OF FUTURES

Futures are almost always useful when you're implementing the client-server model. Any time you have to ask a question of another process, but your program must keep handling incoming messages from other processes, threads and futures are a good approach.

For instance, suppose you've got a pipeline of interconnected processes, where messages are sent from one end to the other. If you're able to run these processes on separate CPUs, you can use threads and futures to make all the processes communicate simultaneously. Every time a process receives an incoming question, have it fork a thread whose job in life is to handle the question. The thread sends the question to the next thread in the chain and waits for the answer. Meanwhile, the process will very happily service other questions. Questions and answers will flow throughout all parts of the pipeline simultaneously. Because the original client can fork many threads, each of which is responsible for a single transaction, there's never any unnecessary delay while a transaction waits for another independent transaction to complete.

Futures are also great in cases where your program must ask questions of two or more processes and then combine the answers in some calculation. Since the order of the transactions isn't critical, you ask all the questions and then use all the answers. When the last answer is in, your program will complete its task. The rule of thumb is that you always ask questions as early as you can and access answers as late as you can. This way, you don't block until all the questions have been asked and all the associated processing has commenced.

in is a future. Of course, unlike the Apple event accessor functions, it never blocks when you call it.

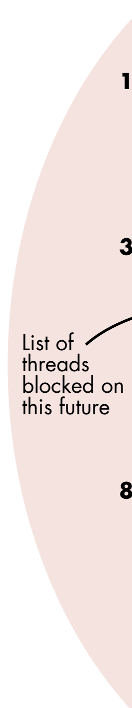
A CLOSER LOOK AT HOW FUTURES WORK

So how do you work with futures in a program? That is, how do you receive a “future” answer to a question and then replace that future with the real answer when it becomes available?

You first prepare your question event just as you would any other Apple event. However, instead of sending out the question with the AESend function, you call the Ask function in the Futures Package. Note that it’s never necessary to install a handler for the answer, as is sometimes the case with AESend. The code that handles the answer is neatly placed after the call to Ask.

Ask returns a fake answer—the future—as a placeholder for the real answer that is to come. Since Ask returns immediately, the client can continue processing. In the meantime, the server has time to receive the question and prepare an answer. The main event loop of the server application receives a high-level event, which it passes to the Apple Event Manager via a call to AEProcessAppleEvent. The Apple Event Manager calls the appropriate event handler routine to receive the question and generate an answer. The client is sent this answer from the server as a normal Apple event reply. Replies that are answers to a future have a special code signifying that they correspond to a particular future. AEProcessAppleEvent recognizes this code and automatically transfers the contents of the real answer into the future. It also calls a hidden routine in the Futures Package that wakes up the threads that are blocked on the future. At this point, the future is no longer a future.

Figure 1 illustrates the sequence of events that are set in motion when a question is asked.



EXAMPLE PROGRAM

The example shown here is a modified version of the TESample program that ships with the Macintosh Programmer’s Workshop. When you start the program, it brings up a window for user interaction as usual. You’ll notice that there’s a new Test menu, with two items that you can choose, Ping and Ping2. First we’ll discuss Ping.

Choose the Ping item to start a conversation with another copy of the program running somewhere on your network. The PPC Browser dialog box appears and prompts you to select another running copy of the program on your machine or elsewhere on the network. The copy you select will be the server; the application that posed the dialog will be the client. At this point you’ll hear some beeps. What this means is that the client is asking the server questions. The server beeps when it receives a question, and then it returns an answer. You can start as many

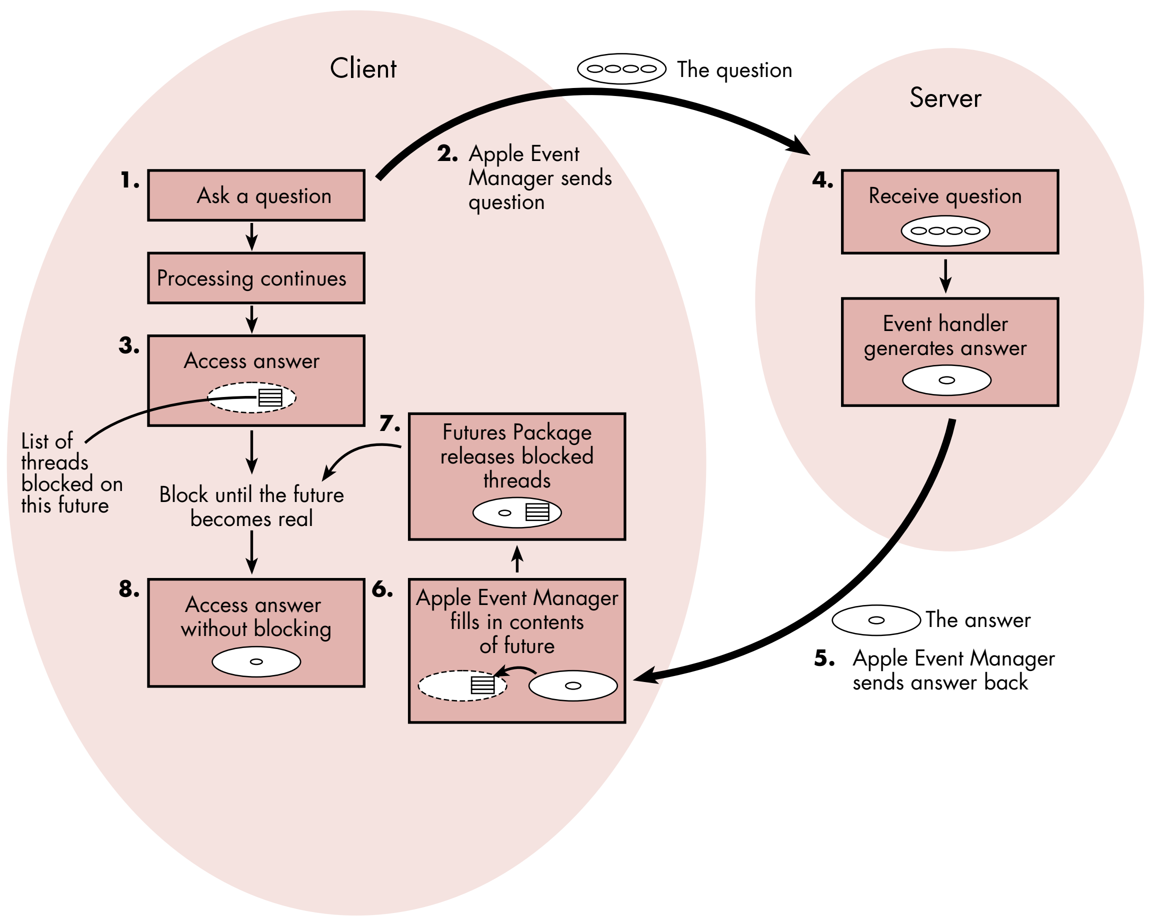

Client
The question
Server
1. Ask a question
2. Apple Event Manager sends question
4. Receive question
Processing continues
3. Access answer
Event handler generates answer
List of threads blocked on this future
7. Futures Package releases blocked threads
Block until the future becomes real
8. Access answer without blocking
6. Apple Event Manager fills in contents of future
The answer
5. Apple Event Manager sends answer back

Figure 1
The Transformation of a Future Into a Real Answer

simultaneous conversations as you like between any two copies of the program. In this way, you establish a number of simultaneous client-server relationships between the various copies of this application. You can even have an application be a client of itself. When this started working for the first time, we had great fun pinging messages around our network (all in the name of testing, of course).

When you select the name of the server from the PPC Browser dialog box, your client program extracts the server's address so that it knows where to direct questions. The client then starts a thread that sends messages to the server. If the

A FEW NOTES ABOUT APPLE EVENTS

To use futures, there are a few things that you need to know about Apple events. The most important thing to be aware of is the AESuspendTheCurrentEvent-AEResumeTheCurrentEvent feature that the Apple Event Manager provides. You need this feature to avoid deadlock when your event handlers ask their own questions. AESuspendTheCurrentEvent allows your event handler to say, "I'm going to return from this handler, but I'm not done processing the event you just gave me. I'll continue processing this event, and I'll call AEResumeTheCurrentEvent when I'm really done."

To use this mechanism in your handler, you suspend the event and fork a thread that asks the question. The original thread that entered the handler returns to the Apple Event Manager with the understanding that the event has not been handled yet. The new thread then constructs an answer to the original question based on the contents of the answer that it received. This may involve copying information from one answer to another. Then the thread calls AEResumeTheCurrentEvent to let the Apple Event Manager know that the event has finally been handled. After calling AEResumeTheCurrentEvent, the thread dies via a call to EndThread.

Without this Apple event mechanism for suspending and resuming events, it would be very difficult to avoid deadlock. Handlers would have to avoid returning because returning implies, "I'm done with this event. It's OK to send the answer back now." Since the handler wouldn't be able to return, the main event loop would be stalled, and deadlock would result. See *Inside Macintosh* Volume VI for more information on the AESuspendTheCurrentEvent-AEResumeTheCurrentEvent feature.

One final note about Apple events: Even though you call Apple event functions in your code to handle certain incoming events, your code isn't truly "Apple-event aware." For your code to achieve this awareness and full System 7.0 compatibility, Apple requires that you implement handlers for standard Apple events like Open Documents and Print Documents. See *Inside Macintosh* Volume VI for details on Apple-event awareness.

answer the client receives is a future, the thread is blocked. When the real answer arrives, the thread is unblocked and is able to access the contents of the answer. After finishing with the answer, the thread properly disposes of both the question and the answer events with the AEDisposeDesc function. The thread then terminates.

Note that the application continues to service the text editing window as usual while servicing incoming requests and driving the conversations it's responsible for. This illustrates that the main event loop is happy and healthy even though many threads may be blocked. Since a thread is not swapped in when it's blocked, a blocked thread has no impact on performance. Practical experience shows that these communications-oriented threads end up spending most of their time blocked.

The second menu item, Ping2, allows you to select two processes. Questions are sent to these processes at just about the same time. This example takes advantage of the asynchrony provided by futures. The first Ask call returns a future. The second Ask is called before the real answer is available for the first call. Then both answers are accessed. This strategy can be used to drive two server CPUs in parallel.

The following code initializes the Threads and Futures Packages and installs an Apple event handler to process ping events. It goes in the main program, right before the main event loop routine is called.

```
#pragma segment Main
main()
{
   •
   •  initialization stuff from TESample
   •

// Initialize the Threads Package.

   InitThreads(false);   // Note the API change.

// Initialize the Futures Package.

   InitFutures();

// Install a handler for the ping messages in the Apple events
// dispatch table, so that when we receive these events, this
// routine will be called.

   AEInstallEventHandler(kSillyEventClass, kPingEvent,
                        (EventHandlerProcPtr) &HandlePing, 0, false);

   EventLoop();             /* Call the main event loop. */
}
```

The following is the handler that processes the ping Apple event. Note that it's in a separate code segment so that procedure pointers to it are jump-table relative.

```
#pragma segment handlers
pascal OSErr HandlePing(AppleEvent question, AppleEvent answer,
                        long handlerRefcon)
{
   char*       stringPtr;
   char        stringBuffer[100];
   long        actualSize;
   DescType    actualType;
   OSErr       theErr;

// Beep to indicate that the question was received.

   SysBeep(120);
// Extract a string from the question.

   theErr = AEGetParamPtr(&question, 'qstr', 'TEXT', &actualType,
            (Ptr) stringBuffer, sizeof(stringBuffer)-1, &actualSize);

// Load a string into the answer.

   stringPtr = "I'm just fine.";
   theErr = AEPutParamPtr(&answer, 'rstr', 'TEXT', stringPtr,
            strlen(stringPtr));
   return(noErr);
}
```

Below is the main event loop. Well, sort of. I cut out some stuff for the sake of brevity. The important thing here is the call to Yield, which gives CPU time to other threads. It's interesting to note that the standard call to WaitNextEvent (not shown) is like Yield in the sense that it gives CPU time to other MultiFinder™ processes. One significant difference is that WaitNextEvent requires that you supply a sleep time of at least one tick when yielding control to other applications. The semantics of the Yield function allow you to regain control as soon as possible, with no obligatory sleep period. The sleep period, as well as the Process Manager's scheduling algorithm, affect the speed with which applications can exchange control and therefore affect the round-trip speed of an IPC transaction.

```
#pragma segment Main
void EventLoop()
{
   •
   •  declaration and initialization stuff from TESample
   •
   do {
      •
      •  Get an event from the event queue and pass it to DoEvent.
      •

// Yield control to other threads.

      Yield();

   } while ( true );  /* Loop forever. We quit via ExitToShell. */
} /*EventLoop*/
```

DoEvent decides what to do with events picked up by the main event loop. Here I've inserted an entry in the case statement that passes high-level events to the Apple

Event Manager routine, AEProcessAppleEvent. Its job is to forward the Apple event to the appropriate handler, in this case HandlePing.

```
#pragma segment Main
void DoEvent(event)
   EventRecord  *event;
{
   •
   • declaration stuff from TESample
   •

   switch ( event->what ) {

// If this is a high-level event, pass it to the Apple Event Manager.

      case kHighLevelEvent:
         AEProcessAppleEvent(event);
         break;
      •
      • Process other kinds of events.
      •
   }
} /*DoEvent*/
```

At this point, we've touched on all of the boilerplate. Now let's take a look at the Ask function at work. Note that a real program would check for errors.

```
#pragma segment Main
void DoMenuCommand(menuResult)
{
   •
   • declaration and initialization stuff
   •
   switch ( menuID ) {
      •
      • Process other kinds of menus.
      •
      case mTest:
         switch ( menuItem ) {
            case iPing:
               {
               OSErr          theErr;
               TargetID       theTargetID;
               PortInfoRec    thePortInfo;
               AEAddressDesc  theAddressDesc;
```

```
                ThreadHandle    theThread;
                AppleEvent      question;
                AppleEvent      answer;
                char*           stringPtr;
                char            stringBuffer[100];
                long            actualSize;
                DescType        actualType;

// Get the target address of the other process.

                theErr = PPCBrowser("\p", "\p", false,
                         &theTargetID.location, &thePortInfo, nil, "\p");
                if (theErr) break;
                theTargetID.name = thePortInfo.name;
                theErr = AECreateDesc(typeTargetID, (Ptr) &theTargetID,
                         sizeof(TargetID), &theAddressDesc);

// Start the thread that pings.

                if (InNewThread(&theThread, kDefaultStackSize))
                   {
                   long i;
                   for (i=0; i<30; i++)
                      {
                      Yield();

// Build an Apple event question that is addressed to the user-
// selected target.

                      theErr = AECreateAppleEvent(kSillyEventClass,
                               kPingEvent, &theAddressDesc,
                               kAutoGenerateReturnID, kAnyTransactionID,
                               &question);

// Load a string into the question.

                      stringPtr = "Hello server, how are you doing?";
                      theErr = AEPutParamPtr(&question, 'qstr', 'TEXT',
                               stringPtr, strlen(stringPtr));

// Ask the question.

                      theErr = Ask(question, &answer);
// If the answer is not a future so soon after Ask, something is
// probably wrong.

                      if (!IsFuture(answer)) Debugger();

// Extract a string from the answer. This will cause the thread to
// block until the answer is received.

                      theErr = AEGetParamPtr(&answer, 'rstr', 'TEXT',
                               &actualType, (Ptr) &stringBuffer,
                               sizeof(stringBuffer)-1, &actualSize);

// If the answer is still a future after you retrieve a string from
// the answer, something is definitely wrong.

                      if (IsFuture(answer)) Debugger();

// Dispose of the answer and the question.

                      theErr = AEDisposeDesc(&answer);
                      theErr = AEDisposeDesc(&question);
                      }

// Dispose of the address descriptor now that the thread no longer
// needs it.

                   theErr = AEDisposeDesc(&theAddressDesc);
                   EndThread(theThread);
                   }
                }
                break;

             case iPing2:
                {
                OSErr           theErr;
                TargetID        theTargetID;
                PortInfoRec     thePortInfo;
                AEAddressDesc   theAddressDesc;
                AEAddressDesc   theAddressDesc2;
                ThreadHandle    theThread;
                AppleEvent      question;
                AppleEvent      question2;
                AppleEvent      answer;
                AppleEvent      answer2;
```

```
// Get the target addresses of the two processes.

                theErr = PPCBrowser("\p", "\p", false,
                         &theTargetID.location, &thePortInfo, nil, "\p");
                if (theErr) break;
                theTargetID.name = thePortInfo.name;
                theErr = AECreateDesc(typeTargetID, (Ptr) &theTargetID,
                         sizeof(TargetID), &theAddressDesc);
                theErr = PPCBrowser("\p", "\p", false,
                         &theTargetID.location, &thePortInfo, nil, "\p");
                if (theErr) break;
                theTargetID.name = thePortInfo.name;
                theErr = AECreateDesc(typeTargetID, (Ptr) &theTargetID,
                         sizeof(TargetID), &theAddressDesc2);

// Start the thread that pings.

                if (InNewThread(&theThread, kDefaultStackSize))
                   {
                   long i;
                   for (i=0; i<30; i++)
                      {
                      Yield();

// Build the questions.

                      theErr = AECreateAppleEvent(kSillyEventClass
                               kPingEvent, &theAddressDesc,
                               kAutoGenerateReturnID, kAnyTransactionID,
                               &question);
                      theErr = AECreateAppleEvent(kSillyEventClass,
                               kPingEvent, &theAddressDesc2,
                               kAutoGenerateReturnID, kAnyTransactionID,
                               &question2);

// Ask the questions.

                      theErr = Ask(question, &answer);
                      theErr = Ask(question2, &answer2);

// Block until the answers become real.

                      theErr = BlockUntilReal(answer);
                      theErr = BlockUntilReal(answer2);
// Dispose of the answers and the questions.

                      theErr = AEDisposeDesc(&answer);
                      theErr = AEDisposeDesc(&answer2);
                      theErr = AEDisposeDesc(&question);
                      theErr = AEDisposeDesc(&question2);
                      }

// Dispose of the address descriptors now that the thread no longer
// needs them.

                   theErr = AEDisposeDesc(&theAddressDesc);
                   theErr = AEDisposeDesc(&theAddressDesc2);
                   EndThread(theThread);
                   }
                }
                break;
```

THE FUTURES API

Here's a description of the routines provided by the Futures Package. Remember that in addition to what you see here, you'll use Apple Event Manager routines to access the contents of a future in the same way that you'd access the contents of any Apple event.

```
pascal void InitFutures ();
```

InitFutures initializes the Futures Package. It lets the Apple Event Manager know that you're using futures. You call it after you initialize the Threads Package.

```
pascal OSErr Ask (AppleEvent question, AppleEvent* answer);
```

You pass in an Apple event question to the Ask function, and it immediately returns a future in the answer parameter.

```
pascal Boolean IsFuture (AppleEvent theMessage);
```

This handy function tells you whether or not a given Apple event is a future, without blocking.

```
pascal OSErr BlockUntilReal (AppleEvent theMessage);
```

This function blocks the execution of the current thread until the specified Apple event is converted from a future to a real answer. If the Apple event is already real, this function returns immediately without blocking. If you ever find yourself accessing a parameter in a future just to cause your thread to block, use this function instead.

THE FUTURE OF FUTURES

Threads and futures make it possible to divide a problem into independent parts that can be executed concurrently. Whenever you can divide a problem into several parts in this way and direct these parts to different CPUs, you can take advantage of parallel processing. One of the most exciting things about threads and futures is, in fact, that they make it very easy to build distributed systems that use multiple CPUs executing in parallel.

Work along these lines is in progress on the Macintosh. The goal is to foster a new era of computing in which users purchase smaller chunks of functionality from a variety of vendors, and then wire them together in new and interesting configurations. Rather than have a single multifunction program, why not have multiple single-function programs that can be spread across several CPUs and seamlessly brought together? In addition to a customized application environment, you also get the advantage of using all the computing power on a network.

Stay tuned for more information about a specific project that makes distributed processing a reality for the Macintosh user.

ACKNOWLEDGMENTS
Thanks to Kurt Piersol for designing the final version of Apple events and for addressing all my concerns before I even posed them. Thanks to Ed Lai, who incorporated support for futures seamlessly into the shipping version of Apple events for me.

Thanks to Larry Tesler for supporting the original version of my “Transporter” research project, which introduced futures to the Macintosh. Larry played a major role in the early brainstorming for Transporter in the Advanced Technology Group, and was the first to relate it to Peter Halstad’s work on futures at DEC. Thanks also to researchers Jed Harris and Joshua Susser, who helped reel in and refine the idea.

Thanks to Laile Di Silvestro, the first person to implement a robust communications timeout mechanism using dreaming threads and futures; to James Redfern, Don Olson, and Eric House for assisting me with past, present, and future versions of Apple events; and to Paul Snively for championing futures in Developer Technical Support.

God bless Geta Carlson, Caroline Rose, Dave Johnson, Monica Meffert, and Louella Pizzuti. You’d be amused to see what these articles start out like, and amazed at how

much editing work goes into them. Special thanks to Don Donoughe, our illustrator. Futures aren't easy to depict, and Don really hung in there as we carved out the figure in this article.

Mondo thanks to Greg Anderson for putting threads and futures into an INIT and for "saving the day" on a daily basis. Thanks to John Wendt for his dedicated work integrating futures with his current secret project. Greg, John, and I are cooking up something really big, so you'd better renew your subscription to *develop*.

WHAT'S HAPPENING WITH THREADS?

The big news is that the Threads and Futures Packages are now shipping together as an INIT. This will allow us to release improved implementations of threads and futures that you can integrate with your code without relinking your application. There is a "glue" object library that replaces the old threads library. Now when you call a routine in the Threads Package, it calls the INIT.

While implementing the "glue" code, we encountered a problem that forced us to change the threads API. First, the InitThreads call now has only one parameter:

```
pascal void InitThreads(Boolean usesFPU);
```

So where did the second parameter go? It's been moved to a new procedure:

```
pascal void SetMainThread(ThreadHandle
   mainThread);
```

This tells the Threads Package that you have a new "customized" thread that you want to use as the main thread. The former main thread dies and is reborn as this new customized thread. The reason for this API change is that the InitThreads glue code now does some initialization that must be done before any thread is created. The old API would have you creating threads before this initialization, which would be bogus.

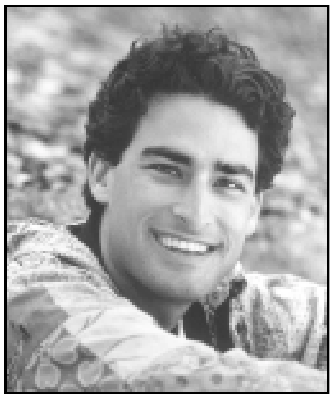
An interesting new feature of the Threads Package is that threads now "dream" when they're sleeping. Each thread has the option of installing a procedure in its fDream field. This routine is pulsed periodically via a call to LetThreadsDream in your event loop. A sleeping thread's dream proc can decide that it has waited too long and call Wake to wake itself from a "nightmare."

Dreaming is extremely useful in implementing robust timeout mechanisms in IPC systems. Instead of using a simplistic timer mechanism, you can place arbitrarily complex logic in the dream proc to decide whether or not your thread should timeout. The dream proc can be designed to be sensitive to periodic messages from a server that say, "I'm still working on your request, so don't timeout." In this way, the dreaming thread knows that it's still in contact with the server, even if it's taking a long time to respond. This approach is far superior to mechanisms that timeout after some fixed period of time.

Thanks to Our Technical Reviewers
Kurt Piersol, Paul Snively, Dave Williams•

USING C++ OBJECTS IN A WORLD OF EXCEPTIONS

The ability to derive C++ objects from MacApp's PascalObject classes yields a powerful marriage, but not without some misunderstandings between the two languages. One potentially thorny area crops up in combining exception handling with dynamic object construction for C++ objects. Judicious use of exception handling techniques can simplify the development and maintenance of robust, well-structured applications. But beware: it's easy to get stuck by the undesired interactions of C++ features and wind up in some tangled brush indeed.



MICHAEL C. GREENSPON

Whereas the C++ language supports dynamic storage management implicitly through object constructors and destructors, Object Pascal relies on user-defined conventions such as those provided by TObject and adhered to explicitly by MacApp®. In addition, MacApp defines conventions for exception handling during object initialization. Here we explore techniques for incorporating MacApp-style exception handling in C++ objects and strategies for object construction, destruction, and dynamic storage management that provide MacApp compatibility. The challenge is to retain the power of C++ features while avoiding some potential pitfalls.

First we examine some basic differences in C++ and Object Pascal semantics and provide an introduction to C++ objects. Then we review the object construction mechanism used by MacApp and by C++, and present techniques for implementing MacApp-compatible exception handling in C++. We also present techniques for using C++ constructors and destructors with PascalObject-derived classes. Finally we explore some special difficulties and workarounds for using C++ member objects in handle-based classes.

NOT ALL OBJECTS ARE CREATED EQUAL

Both C++ and Object Pascal rightfully claim to be "object-oriented" languages; yet, in fact, there are some fundamental differences in expressiveness and meaning

MICHAEL GREENSPON is the principal noisemaker for Integral Information Systems, a Berkeley, California software engineering and consulting group (AppleLink: Integral). When he's not breaking compilers by trying to use all of their features at once, he's busy designing next-generation solutions for clients. His interest in the evolution of information systems goes beyond silicon—as a neurobiology undergrad at Cal Berkeley he developed visualization tools for neural network dynamic modeling using a Macintosh workstation linked to the school's Cray supercomputer. "People think the brain's a computer, but it's really an aquarium." (Ask him about his lava lamp representation of the mind.) A native Californian, he says he "prefers UV to ELF, any day." In fact, when the sun's out you're likely to find him swimming, mountain biking in between seemingly similar constructs in the two languages. These differences can be seen by comparing the object creation process in the two languages:

```
{ Object Pascal: }
Var obj: TObj;
   Begin
      New(obj);    { Allocate heap storage for a TObj instance }
   End;

// C++:
TObj* obj = new TObj;     // Allocate and construct a TObj instance on
                          // the heap
```

PASCAL NEW STATEMENT ALLOCATES STORAGE

The Pascal New statement allocates relocatable storage on the heap and places a reference to the storage in the declared object reference variable. In the MacApp environment, all Pascal objects are allocated as relocatable heap blocks using NewHandle, so the reference is a Memory Manager handle. In the example above, executing the Pascal New statement does not provide a fully constructed object instance, but merely initializes the storage enough to give the object its class identity. To become a true object instance, the storage must be initialized explicitly by the programmer. By convention in MacApp, this is done by calling the method I«Classname», where «Classname» is the class of the object being instantiated—IObj, for example. Referring to object fields through the object reference variable before this explicit initialization will probably yield garbage results, greetings from Mr. Bus Error, or worse.

C++ NEW STATEMENT INSTANTIATES OBJECTS

In contrast, the C++ compiler gives a passing **new** statement much deeper consideration. In general, a C++ compiler translates the **new** statement into a call to a **new** operator followed by a call to a constructor for the class. The **new** operator is similar to the New statement in Pascal: it's a function responsible for allocating storage for the object instance. A constructor is a function responsible for changing that raw storage into an instance of the class—a fully constructed object. Both operator new and a default constructor are provided by the language system or generated by the compiler and may be redefined, overloaded, and overridden for each class by the user.

C TRANSLATED CODE

MPW C++ and other C++ systems based on the CFront translator instantiate objects by generating a single call to the appropriate constructor. This constructor calls operator new explicitly and, if the allocation is successful, constructs the object. For a class with a trivial user-defined default constructor, the generated C code looks like this (assuming the TObj class is derived from a HandleObject):

Wildcat Canyon, or backpacking in the High Sierra. In between, he's working to promote telecommuting, car-free days, and CRT-free spaces.•

```
struct TObj** obj = __ct__4TObjFv( 0 );      //  TObj* obj = new TObj;
```

The trivial constructor itself is spit out by the translator as

```
// Translation of definition TObj::TObj() { /* user code goes here */ }
struct TObj** __ct__4TObjFv(struct TObj** this) {
   if (this || (this =
      (struct TObj**)__nw__12HandleObjectSFUi(sizeof(struct TObj)))) {
   // A nontrivial constructor would have user code here
   }
   return this;
}
```

If you can squint past the mangled function names, you'll see that the **new** statement has been translated as an explicit call to the default constructor for the class. The constructor is named __ct__4TObjFv, which loosely unmangles as "a constructor function for class TObj taking void (no) arguments." This default constructor (there can be multiple overloaded constructors for a class) is called without user-supplied arguments. That's how we declared it, but the translator snuck in another argument—the **this** reference for the object being constructed. The constructor is passed a nil **this** reference, indicating that no storage is allocated and the constructor needs to do so by calling operator new.

Looking at the translation of the constructor itself, you'll see that the code tests the **this** reference and, if it's nil, then calls the function __nw__12HandleObjectSFUi (operator new function for class HandleObject taking an unsigned int argument). The intrinsic HandleObject::operator new just calls NewHandle unless overridden. If operator new returns a non-nil value, the constructor assumes it's a reference to storage for the nascent object and executes its body of initialization code.

We'll come back to constructors and translations when we explore constructor implementation for objects in a multilevel class hierarchy. A clear understanding of the object instantiation process is needed to use exception handling with dynamic storage management. For now, just notice that Pascal's New statement provides storage, while C++'s **new** statement provides objects.

C++ OBJECT STORAGE CLASSES

In MacApp, Pascal objects are allocated only as handles on the heap. In C++, however, you can specify several ways for the compiler to get storage for an object:

- dynamically on the heap with operator new
- automatically on the stack in the scope of any code block
- in the static data space, courtesy of the linker

Reading the intermediate C code can save you a lot of MacsBug time. The C code can be dumped into a file by using the –c option on the compilation command line and redirecting the output with > *file* (for example, cplus foo.cp –c –l0 > foo.c). Including the –l0 (el zero) option prevents the generation of #line directives.•

For convenience we'll distinguish between *heap-based* objects that are only accessible by pointers or handles and auto and static objects, which we'll call *stack-based.* Regardless of where an object is allocated, a constructor is invoked to create the instance. This is also true for temporary objects generated by the compiler under certain circumstances (such as for function arguments and return values). A constructor is always invoked when a new class object is created, no matter where or how. This is one of the desirable properties that makes all C++ objects first-class citizens regardless of storage class.

STATIC POTENTIAL

The low run-time overhead needed to allocate storage for stack-based objects makes them light and suitable for implementations where heap-based objects would be too inefficient. Further, many objects have limited enough scope to be allocated on the stack. Thus the ability of C++ to create stack-based first-class objects is a powerful feature that allows us to make our programs more intensively object-oriented than we could with only heap-based objects.

For example, consider a dynamic list class like MacApp's TList. A TList is a Pascal heap object that dynamically adjusts its size to store (typically) handles to other Pascal heap objects. This is a good arrangement for relatively short lists of things like windows, documents, and views. But what if you want a list of 50,000 objects that are small in storage but complex enough operationally to be encapsulated as a class?

Such small but complex objects are common and include things like atoms, strings, and lists themselves—the nuts and bolts of data structures. Each of these classes owns and manages dynamic storage, yet instances of them rarely need to be allocated on the heap. For example, a string class in C++ can be represented statically as a handle to storage for characters; yet it can have many operators (concatenation, equality, assignment) defined for it and a constructor and destructor that conspire to count references and perform garbage collection on the handle. Just treating a string as a typedef synonym for a handle would lose the encapsulation and notational convenience of the operator family. A string that derived from a TObject (or other heap-based object) would still be first-class, but doing something simple, like building a long TList of strings, could get pretty inefficient.

A possible C++ solution would be to define a StaticList class that manages a single handle to storage for a list of small static objects, such as strings. Having the list class manage the allocation of objects in the list (which themselves manage handles to storage) rather than allocating them as heap-based objects reduces the memory management overhead for the list by more than half.

However, until the addition of language and compiler support for exception handling, throwing exceptions from stack-based object constructors is basically a no-no, because the compiler doesn't generate the needed destructor call. Stack-based objects are useful anyway, even though their constructors can't throw exceptions; and

For an example of a garbage-collecting string class see "The Power of C++," by Waldemar Horwat, MacHack Conference Proceedings, 1990.•

it's useful to throw exceptions from heap-based object constructors, as MacApp does for its Pascal classes. We're going to take a closer look at how exceptions and construction interact in a moment, but first let's consider member objects.

MEMBERS ONLY

In addition to the heap-based, auto, and static storage classes described above, C++ objects can also be allocated as members of an enclosing object. This is different from the common practice of maintaining an object field with a reference (such as a handle) to another heap-based object—an owned object allocated separately from the owning object. Members take their storage from the storage of their enclosing object. Like all C++ objects, members are first-class in that they can have constructors, destructors, and all other class properties. Member objects are a powerful feature of C++, but using them in a handle-based world and in the presence of exceptions presents some special difficulties. We'll examine these difficulties later and for now restrict the discussion to heap-based objects *without* member objects.

EXCEPTIONS DURING OBJECT CONSTRUCTION

MacApp and C++ each provide a functionally similar scheme for object instantiation, and it's important to understand these mechanisms in order to use exception handling during object construction. During the instantiation process, object fields must be initialized to a known state before any failures can occur. Then, if a failure does occur, the exception handler can safely destroy the partially constructed object and free its storage.

THE MACAPP WAY

After the Pascal New statement allocates uninitialized heap storage, an explicit initialization step is required to instantiate an object in the allocated space. In the MacApp environment, the root TObject class provides the canonical framework for this initialization. Figure 1 illustrates this initialization process, showing the flow of control during construction of a three-class Pascal object hierarchy.

THE C++ WAY

C++ provides implicitly in its language semantics an instantiation scheme functionally similar to MacApp's conventions. When the CFront translator generates code for a derived class constructor, it automatically inserts calls to the base class constructors before executing the user-supplied body. Figure 2 shows an overview of the C++ object instantiation process.

There are several things to notice in Figure 2:

- The class hierarchy is derived from class HandleObject, which is a native C++ class in that it uses the C++ virtual function table dispatching mechanism.

For more information on object initialization in MacApp see the *MacApp 2.0 Cookbook, Beta Draft*, APDA #M0299LL/C, Chapters 1 and 7.•

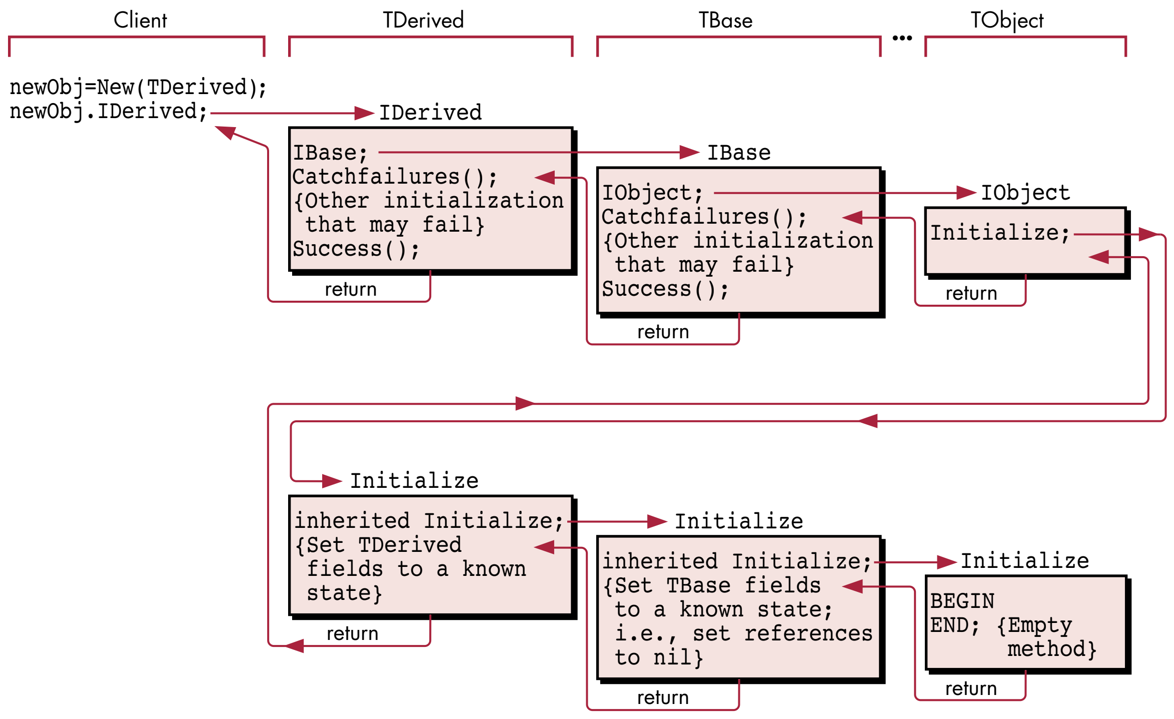


Figure 1
Flow of Control During Pascal Object Instantiation

- Each derived class constructor calls its immediate base class constructor before executing its own body so that the base portions of the object are constructed before the derived portions. The example class hierarchy has class CDerived descended from CBase descended from HandleObject.
- The virtual function table pointer (vptr) for the object being constructed is initialized to the constructor's class. This narrows the object's type to that of the constructor's class. So although we may be constructing an object of class CDerived, within the constructor CBase::CBase the object is essentially one of class CBase with respect to virtual function calls.

The user-defined body of each constructor initializes the fields belonging to that class and performs other constructions, such as allocating owned objects. The narrowing of type in constructors is important for exception handling, because if a constructor signals a failure, we want to delete the partially constructed object using a virtual destructor. Without narrowing, the virtual destructor call would resolve to the most-derived class's destructor. This destructor would expect to operate on fields

For more information on object initialization in C++ see *The Annotated C++ Reference Manual*, by Margaret A. Ellis and Bjarne Stroustrup, Addison-Wesley, 1990, §12.6.2 (on base/member initialization syntax) and §10.9c (on virtual function instantiation).•

of the derived object, which has not yet been constructed—Heap Check time! With narrowing, the destructor invoked is of the same class as the constructor signaling the failure, and only the constructed portions of the object are destroyed.

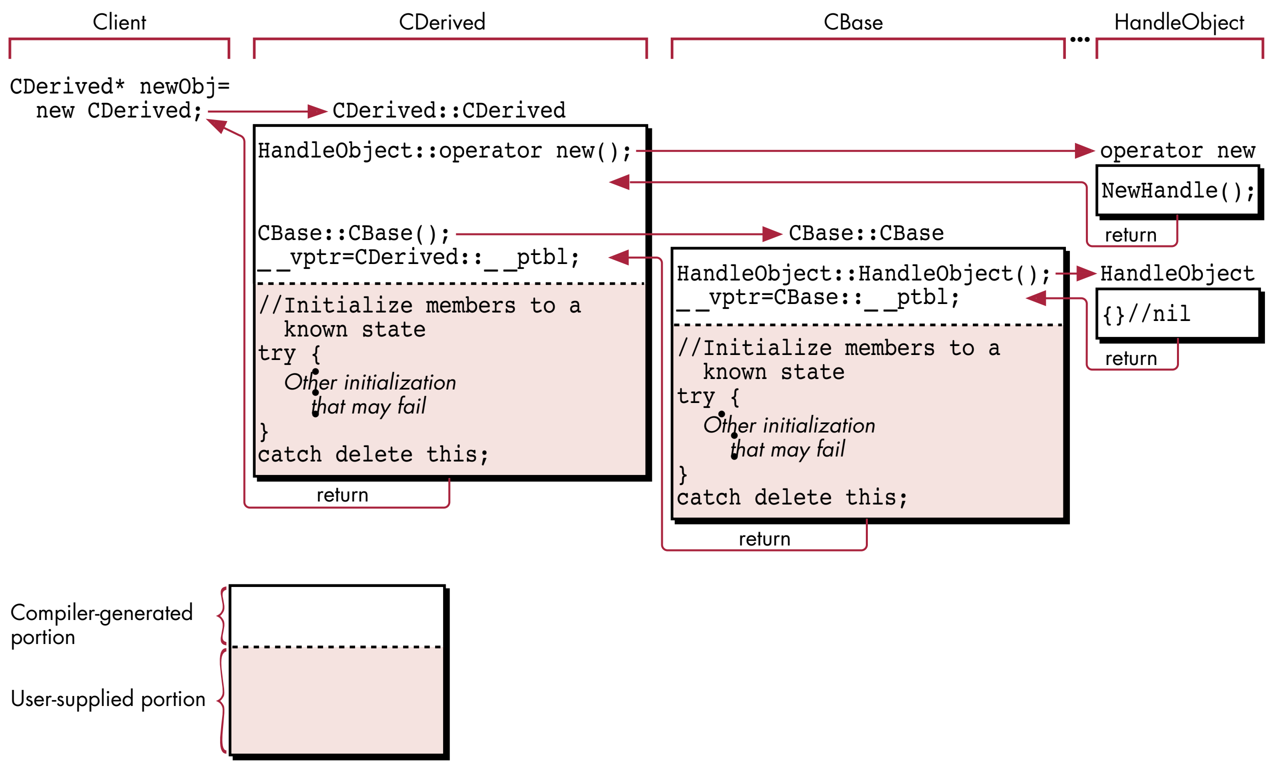


Figure 2
Flow of Control During C++ Object Instantiation

Since the constructors for each class in the hierarchy typically set fields to 0 in order to initialize the object to a freeable state, sometimes it's more convenient and efficient to initialize the entire block of storage to 0 when it's allocated. One way to do this for native C++ objects is by redefining operator new. (See "ClearHandleObject Approach for Native C++ Objects.")

EXCEPTIONS IN CONSTRUCTORS

To implement a simple but convenient MacApp-compatible failure handling mechanism in C++, we use the **try** and **catch** macros (see "Exception-Handling Macros"). In general, if a constructor does anything that could throw an exception (say, due to failure of an owned object allocation), it must take responsibility for catching all exceptions and deleting its object on failure. If the constructor doesn't

CLEARHANDLEOBJECT APPROACH FOR NATIVE C++ OBJECTS

Here's a simple technique that redefines operator new for HandleObject to call NewHandleClear, so that on entry to derived constructors, all object fields are initialized to 0 (references to nil, flags to false). This saves you the trouble of explicitly initializing these fields to 0 with constructor code before doing anything that could fail.

```
class ClearHandleObject : public HandleObject {    // Provides HandleObjects pre-
                                                   // initialized to zero
   public:
               void** operator new(size_t theSize) { return NewHandleClear(theSize); }

// Could declare some other useful handle-oriented functions here (or in a superclass)
//             Boolean   Lock(Boolean);        // Lock/unlock object and return previous
                                               // state, like TObject
//             void      MoveHigh();           // Move object handle out of the way to top
                                               // of heap
//             etc.
```

delete its object on failure, no other code will have the **this** reference to do so. This is also true for a hierarchy of constructors—each base constructor must catch all exceptions it could generate and delete its object on failure before throwing to the next handler. In a typical case, each constructor/destructor pair might look like this:

```
CDerived::CDerived : CBase(...), fOwned(nil) (...) { // Reference is nil to start
   try {
      fOwned = new TOwned;
      // Other stuff that could fail
   }
   catch delete this;    // If failure, destroy and throw to next handler
}

CDerived::~CDerived() {   // Virtual destructor
   if (fOwned) delete fOwned;
   // Other cleanup
}
```

USING CONSTRUCTORS AND DESTRUCTORS WITH PASCALOBJECTS

As C++ programmers, we'd like to use constructors and destructors with PascalObject-derived classes to attain a uniform interface for clients. That is, in C++ programs we'd like to instantiate all objects, whether native or PascalObject-derived,

For more information on proposed C++ exception handling see *The Annotated C++ Reference Manual*, by Ellis and Stroustrup, §15.•

EXCEPTION-HANDLING MACROS

Here we present a derivative of Andy Shebanow and Andy Heninger's excellent UFailure-compatible exception handling scheme for C and C++ that was distributed with Sample Code #14, CPlusTESample, on the *Developer Essentials* disc. The basic scheme is the same, but we've tweaked the macros so that they follow C block structuring conventions more closely. This produces code that's easier to read and more compatible in form with proposed C++ language extensions for exception handling. Now we can write:

```
try {
   // stuff that might throw an exception
}
catch {
   // do stuff to recover
   break;    // Exit handler, recovered
}
```

This establishes a failure handler within the scope of the **try {}** block. If any code within this block generates an exception (by calling Failure) the exception will be caught by the code in the **catch {}** block. Since we're following C structuring conventions, for simple statements you can omit the {}.

Normally, falling through the end of the **catch** block will throw the exception to the next handler in the chain by calling Failure again. If you want to recover from the exception, you can execute an explicit goto out of the catch or simply execute a **break** statement. For example,

```
try CouldFail();
catch break;
```

would recover from all exceptions in CouldFail without further checking. Also, you can do a **break** in a **try** block that just exits the try. The **catch** block is executed only if something in the **try** block throws an exception.

The following caveats apply:

- You can only have one try/catch pair per code block. If you want more than one, enclose each try/catch pair in its own block. But you'll find that when you have multiple **try** blocks in a function, they often occur within a block already (if, while), so this is not really a big deal.
- CFront is thought to have problems with macro expansions in constructors. Sometimes macro expansions are positioned incorrectly relative to code the compiler inserts for calling operator new and base class constructors! If you're getting weird results, be sure to look at the generated C code to see what's really going on. As MacApp would say, "You Are Warned"—which makes it OK, right?

MPW C does a good job of optimizing out the loops, making the generated code comparable to the previous versions.

Here are the macro definitions to replace the previous version:

```
#define try \
   jmp_buf errorBuf; \
   if (! setjmp(errorBuf) ) { \
      FailInfo  fi; \
      CatchFailures(&fi, StandardHandler, \
         errorBuf); \
      do {

#define catch \
      } while (0); \
      Success(&fi); \
   } \
   else \
      for(; (1); Failure(gFailError, \
                     gFailMessage))
```

VIRTUAL DESTRUCTORS AND PASCALOBJECTS

MPW C++ 3.1 uses a Pascal method dispatch instead of a static call for the base class destructor calls it generates at the end of a derived destructor that is declared virtual in a PascalObject-derived class. The Pascal method dispatch resolves (virtually) to the most-derived class's destructor, which is the caller—in other words, death by infinite recursion. Oops.

Without virtual destructors, the **delete** statement won't operate polymorphically. In other words, you can get bitten by this:

```
funfun() {                         // TBase has a nonvirtual
                                   // destructor
   TBase* anObj = new TDerived;    // Create a new TDerived
   delete anObj;                   // But delete a TBase! Ugh!
   delete (TDerived*) anObj;       // This works correctly but what
                                   // a pain--error prone, too
}
```

Until the C++ compiler is updated, you should adopt the convention of using the Free method as the virtual destructor chain, and redefine PascalObject::operator delete to invoke it.

with a **new** statement and destroy all objects with a **delete** statement. In general, we expect objects to follow language semantics, regardless of their storage class or implementation, and we want to encapsulate the MacApp initialization scheme so that our C++ clients don't have to know its details.

This seems straightforward, but there are several difficulties in using constructors and destructors with C++ classes derived from PascalObjects. Though handle-based, these classes use the Pascal method dispatcher and not the C++ virtual function table mechanism. Thus, the native C++ narrowing isn't generated in constructors, making method calls (such as Free) always virtual to the most-derived class—even if that part of the object hasn't been constructed yet. Further, as of the E.T.O. #3 release (C++ 3.1) there's a problem with how MPW C++ translates virtual destructors in PascalObject hierarchies (see "Virtual Destructors and PascalObjects").

These considerations suggest some guidelines for those determined to make use of object constructors and destructors to attain some consistency between native C++ objects and descendents of PascalObjects:

- For C++ classes descended *directly* from MacApp classes, define constructors CObj::CObj(...) as you would a MacApp IType method. That is, the constructor should initialize fields of the object to a known state and then call inherited::IType before performing any additional construction that could fail—for example, before allocating owned objects.
- For C++ classes descended *indirectly* from MacApp classes, the compiler invokes the base class constructors (which can fail) *before* the constructor body can execute to set up a handler. Therefore, in order for your Free method to operate correctly, you *must* define a virtual pascal function Initialize to initialize fields to a known state. MacApp calls this method before doing anything that could fail. Your constructor can simply call inherited::IType before performing any initialization that could fail.
- If you call any other virtual member functions in your constructors, make sure that the fields they depend on are initialized by your Initialize method. Virtual function calls in PascalObject hierarchies are always instantiated as their most-derived definitions, even before derived constructors are executed to construct the derived parts.
- Constructors should perform operations that could fail within the scope of an exception-handling **try** block. The **catch** block should, if it can't recover from the exception, perform any special cleanup and then delete the partially constructed object by executing the statement **delete this**.

For destruction, we need a workaround because the native virtual destructor mechanism is inoperative in PascalObject hierarchies. We would like the **delete** statement to invoke the Free method chain, which functions as the canonical Pascal virtual destructor (TObject::Free deletes the storage). Here's one possible solution that may require minor revision with future releases of MacApp:

- Do not define any destructors in derived C++ classes. Define a virtual pascal function Free to perform the cleanup functions you would have put in a virtual destructor and then call inherited::Free. This method then becomes the canonical virtual destructor and allows the object to be destroyed and disposed of by Pascal code.
- In order for C++ code to destroy and free the object with a **delete** statement, redefine PascalObject::operator delete to invoke Free:

```
void PascalObject::operator delete(void** h) {
    ((TObject*)h->Free(); // Invoke canonical virtual
                          // destructor chain
}
```

Future versions of MacApp may rely on operator delete to actually dispose of the object's storage; if this becomes the case, you can modify TObject::Free or TObject::ShallowFree to do the right thing.

If you're using member objects with destructors in PascalObject hierarchies, there are other problems, as discussed in the next section.

IT'S DIFFERENT WITH MEMBERS AND AUTOS

So far we've been discussing classes that do not contain member objects. This covers all standard MacApp classes, provided your derivatives don't add members. To take advantage of first-class member objects, which are a powerful feature of C++, we must face some difficulties. There are problems using member objects with handle-based classes (both HandleObjects and PascalObjects) and problems with exceptions in member constructors (common to static/auto objects as well). There are further complications with PascalObjects due to the lack of virtual destructors.

Nevertheless, on balance we think member objects represent a powerful enough construct to justify exploring these problems and possible workarounds—if we can't have the compiler support that's really needed. The following sections offer techniques for overcoming problems encountered with member objects in regard to handle-based classes and exception handling.

Declare a wrapper class for member objects in handle-based classes.
Consider this translation of a constructor for a handle-based class that has a member object with a constructor:

```
struct foo : public PascalObject {  foo();  Memb  a;  /* a member object */ };
foo::foo() {}             // Default constructor

// Translation of foo::foo()
struct foo** __ct__3fooFv(struct foo** this) {
   if (this ||
      (this = (struct foo**)__nw__12PascalObjectSFPFv_vUi(_foo, sizeof(struct foo)))) {
         __ct__4MembFv(& (*this)-> a) ;  // Compiler-generated call of member constructor!
   }
   return this;
}
```

See the problem? It's the **this** reference passed to the member object's constructor behind our backs by the compiler—it's a dereferenced handle! If the member constructor does anything interesting (like allocate memory) it could move the enclosing object, leaving the code with a dreaded dangling pointer. This is true for all nonstatic member functions of member objects (not just constructors). Danger, Will Robinson!

A general workaround involves locking the enclosing object's handle before calling member functions that can move memory—the question is who should do the locking. If you have a library of classes you'd like to use as members in handle-based objects, you may want to create wrapper declarations for these classes and pass the handle to be locked (the **this** reference of the enclosing object). Here's an example of a wrapper for use by TObjects (which have a Lock method built in). It looks like a lot of code, but it's all declarations. The run-time overhead is negligible—a trade-off between using HLock/HUnlock to make the member safe and having a separately allocated heap object.

```
class CObj {    // Some library class we want to use as a member in handle-based objects
   public:
                  CObj(...);                  // Constructor, could move memory
   virtual int  Accessor() { return field; }    // Won't move memory
   virtual void Funky(...);                   // Could move memory

   private:  int   field;
};

// A utility class with a constructor that locks handles
class Lockit {     // Lock the enclosing handle in constructor chain before member
                  // constructor is called
   public:
                  Lockit(TObject* h) { h->Lock(true); }     // Lock the handle
// This unfortunately defines a 1-byte structure rather than zero-length
};

// A wrapper for the above functional class CObj——reexport via private base class.
// Also inherit from Lockit so that its constructor is called before CObj::CObj().
// Member functions and locking wrappers must be inline or in a resident segment. Otherwise,
// calling these functions can trigger a segment load and heap scramble before we can lock
// the enclosing object.
class MObj : private Lockit, private CObj {     // Wrapper for using CObjs in handle-based
                                                // classes
   public:
// Provide handle-locking wrappers for functions that can move memory
               MObj(TObject* h,...);
//             MObj(HandleObject* h,...);         // Could overload all to work with
                                                   // HandleObjects too
   virtual void Funky(TObject* h,...);
// Now we'd like to use the access declaration syntax to republicize functions that don't
// move memory, but unfortunately CFront currently miscalculates the 'this' reference! To
// get 'this' right, we have to provide an explicit inline call, which is messy in this
// declaration but doesn't add any run-time overhead.
//             CObj::Accessor;             // Doesn't work, miscalculates 'this'!
// Workaround:
   virtual int  Accessor() { return CObj::Accessor(); }
                                           // Inline call wrapper so 'this' is right
};

// Wrapper function for constructor--lock enclosing handle first
inline MObj::MObj(TObject* h,...) : Lockit(h), CObj(...) {
   h->Lock(false);    // Unlock enclosing handle now that we've finished with base constructors
}

inline void MObj::Funky(TObject* h,...) {    // Some memory-moving member function to wrap
   Boolean state = h->Lock(true);            // Lock the handle--preserve its previous state
   CObj::Funky(...);                         // Call original function
   h->Lock(state);                           // Put handle back the way it was
}

// Clients can use CObj as a wrapped member like this:
class TFoo : public TObject {
         MObj     fMember;                   // Include the wrapped member
   public:
                  TFoo(...) : fMember(this,...), ... { ... }   // Be sure to pass 'this' to
                                                               // member constructor
   virtual void   Func() { fMember.Funky(this,...); }          // Pass 'this' to member
                                                               // functions for HLocking
};
```

Notice that this wrapper scheme has some drawbacks—for example, the requirement of explicitly passing a **this** reference as an additional argument. This won't work for functions such as operator functions that have a fixed number of arguments. Similarly, there's no way to pass an explicit argument to a destructor. In these cases, you can get by if you don't refer to any member object fields within member object functions after doing anything that can move memory. For example:

```
CObj::~CObj() {     // Destructor for above example library class that's used
                    // as a member in handle-based objects
   // Can use our fields here; our 'this' reference is a dereferenced handle!
   delete fOwned;        // Dispose of some storage we were managing--could compact heap?
   // Better not reference any fields here! Our 'this' reference could now be a dangling
   // pointer!
}
```

For more information on member access declaration syntax see *The Annotated C++ Reference Manual*, by Ellis and Stroustrup, §11.3.•

If you can't guarantee not referencing member fields after doing anything that can move memory, then you'll have to explicitly lock and unlock your enclosing objects before calling member object functions.

Don't throw exceptions from member/auto object constructors. Because the compiler invokes member object constructors *before* the body of the calling constructor can execute to set up an exception handler, it's a bad idea to throw exceptions from member and auto object constructors. If the member constructor throws an exception, it's caught outside the scope of the calling constructor. The object is only partially constructed, but fully allocated, and no code has the **this** reference to delete the storage.

Therefore, member objects should *not* throw exceptions from their constructors. For similar reasons, it's inadvisable to throw exceptions from constructors for classes used as auto objects. The exceptions are caught by a calling function in a higher stack frame, and other autos in the original frame aren't destroyed correctly.

Explicitly test successful initialization of members/auto objects. To deal with member and auto objects with constructors that may fail, and to avoid memory leaks and worse, we really need language and compiler support for exception handling. Such support has been proposed for a while, but it may be a long time coming. In the interim, we'll fill in with conventions and guidelines for member and auto objects that manage storage and perform other operations in their constructors that may not succeed.

Possible conventions include explicitly initializing instead of using constructors (which we've been trying to avoid) or explicitly testing for successful initialization (which we prefer). A nice way to test explicitly is to define operator! as a test for failure. This convention follows the C notion of using ! to test for a nil pointer indicating an allocation failure. For example, consider a constructor for a class that has a member object:

```
TObj::TObj() : memb(this,...) {      // Initialize members with member
                                     // initialization syntax
  try {
     if (!memb) Failure(err,msg);    // Test explicitly for member
                                     // initialization failure
     // Do other construction that could fail
  }
  catch delete this;                 // Destroy object if failure occurs
}
```

Here's code for the member class constructor and operator! :

```
TMemb::TMemb(...) {
   fOwned=nil;
   try fOwned=new TOwned;  // Don't throw exceptions if failure occurs
   catch break;            // Just exit from handler chain--i.e., recover
}

TMemb::operator!() {
   return fOwned==nil ||
      (other init failure);  // Return *true* if initialization failed
}
```

Call destructors explicitly for auto objects in exception handlers. A final convention for using auto objects allocated within code blocks that can generate exceptions (either themselves, or by calling things that can fail) is to explicitly destroy these autos in your exception handler. This can be done by calling the destructor function directly using the static call syntax obj.TObj::~TObj().

Normally, the compiler knows to destroy autos when they go out of scope. Unfortunately, the compiler doesn't yet know about exceptions and stack unwinding, so it doesn't know that a call to Failure is blasting us out of scope. This can cause memory leaks and worse, so always be careful with auto objects that manage storage in the presence of exceptions. In particular, don't declare autos that require destruction within the scope of a **try** block. For example:

```
MapFile(TFile* aFile) {
   String s(10000);                 // Construct a 10K dynamic string on the heap
   // Be sure to catch all exceptions now so we can free our autos
   try {    // Don't declare autos needing destruction within this block!
      FailInit(!s);                 // Throw memFullErr if initialization failed
      aFile->ReadIntoString(s);     // Do our work--could fail
   }
   catch s.String::~String();       // Destroy auto explicitly and throw to next handler frame
}
```

Be aware of implications of no PascalObject virtual destructors for members. Previously we recommended using operator delete to invoke Free instead of defining destructors in PascalObject hierarchies. But, if you include member objects with destructors in your classes, the compiler generates calls to these destructors *before* Free is invoked. That is, *all* member objects in the hierarchy are destroyed before any of their enclosing objects are destroyed. This is not a problem as long as your Free methods don't try to access the member objects. Finally, because there's no declared virtual destructor, the **delete** statement won't operate polymorphically with respect to the members. Be sure to delete the derived class and not a base class.

AN EXAMPLE

Finally, here's some sample code that illustrates all the techniques mentioned above:

```
// A fictitious example illustrating the techniques described above
// Let's declare this mess once and for all
typedef pascal void (*DoToField)(StringPtr fieldName, Ptr fieldAddr, short fieldType,
                      void *DoToField_StaticLink);

class Lockit;                                     // Declared earlier in article
inline void FailInit(Boolean t) { t? Failure(memFullErr,0) : ; }

class TMyEvtHandler : public TEvtHandler {        // Derived directly from a MacApp class
   public:                   // Constructors and destructors
                             TMyEvtHandler(TMyCommand* aCmd=nil,TMyDocument* aDoc=nil);
//                           ~TMyEvtHandler();      // No C++ virtual destructors for
                                                  // PascalObjects yet!
                                                  // Override operator delete instead
      virtual pascal void    Free(void);          // Canonical Pascal virtual destructor

                             // Other methods
#if   qInspector
      virtual pascal void       Fields(DoToField, void* DoToField_StaticLink);
#endif
      virtual pascal Boolean    DoIdle(IdlePhase idlePhase);
   // ...
   private:
      TMyDocument*     fDocument;       // References to objects owned by someone else
      TMyCommand*      fCommand;
      TMyOwned*        fOwned;          // Reference to an object we own
      MStaticList      fStringList;     // A wrapped member object--must take special care
                                        // in its destructor to not access fields after
                                        // moving memory
};

class TMySpecialEvtHandler : public TMyEvtHandler {      // Derived indirectly from a
                                                         // MacApp class
   public:                         // Constructors and destructors
                             TMySpecialEvtHandler();     // Default constructor
      virtual pascal void    Initialize(void);           // Pascal-style constructor
                                                         // No C++ destructor
      virtual pascal void    Free(void);                 // Pascal-style virtual destructor

                                   // Other methods
};
```

```
// Use member initialization syntax to pass arguments to our member object constructors
TMyEvtHandler::TMyEvtHandler(TMyCommand* itsCmd,TMyDocument* itsDocument) :
      fStringList(this,sizeof(String),String::CopyConstructor,String::~String) {

   // Initialize fields to a known state
   fDocument = itsDocument;
   fCommand = itsCmd;
   fMyOwned = nil;                    // So we won't try to delete it until we allocate it!

   // Call IType chain to init MacApp classes
   IEvtHandler(nil);                  // Initialize inherited MacApp classes

   try {     // Do rest of initialization in the context of a failure handler
      // Make sure our member objects initialized themselves OK
      FailInit(!fStringList);         // Make sure MStaticList member was initialized OK

      // Do the rest of our construction (e.g., allocate owned objects)
      FailNIL(fMyOwned = new TMyOwned);            // . . .

      // Do anything else that could perhaps fail
      gApplication->InstallCohandler(this,true);   // Install ourselves in the idle chain
   }
   catch delete this;                 // Oops, something failed, so just Free ourselves
}

pascal void  TMyEvtHandler::Free(void) {
   gApplication->InstallCohandler(this,false);  // Get us out of the idle chain
   DeleteIfNotNil(fMyOwned);          // Delete our owned objects if they were allocated
   try fDocument->Notify();           // Try something that could fail
   catch break;                       // Just recover--destructors can't throw exceptions!!
   inherited::Free();                 // Tell bases to destroy themselves
                                      // TObject::Free will delete the storage
}

// Use base initialization syntax
TMySpecialEvtHandler::TMySpecialEvtHandler() :  // Derived indirectly from a MacApp class
      TMyEvtHandler(gSomeCommand,gSomeDoc) {

   // By the time we get here, all base and member constructor and IType chains have executed
   // without failure

   try {                              // Initialization that could fail
   }
   catch delete this;
};
```

```
pascal void TMySpecialEvtHandler::Initialize(void) {
   inherited::Initialize();          // Initialize inherited fields to a known state
   // Initialize our fields to a known state here!
}

pascal void TMySpecialEvtHandler::Free(void) {
   // Delete owned objects and perform other cleanup here
   inherited::Free();                // Let base classes destroy themselves
}
```

SUMMARY

Here's a summary of the various techniques we've discussed for successfully using C++ objects in a world of exceptions:

- Use exception-handling macros to make code more reliable and easier to maintain.
- Throw exceptions from heap-based object constructors only after deleting the object being constructed.
- Don't use destructors in PascalObject-derived hierarchies; instead redefine PascalObject::operator delete to invoke the Free chain.
- Don't throw exceptions from member object or auto object constructors; instead use explicit tests of successful initialization (such as operator!).
- Remember to explicitly destroy auto objects before exiting their scope by throwing an exception.
- Use a wrapper class or other technique to lock enclosing object handles when using member object functions that can move memory within handle-based objects.
- If you're brave enough to use members with destructors within PascalObjects, be aware of the implications of not having virtual destructors.
- When in doubt, compile with –c –l0 and check the translated C code.

CONCLUSION

As an evolutionary outgrowth of the C language, C++ adds the basic features needed to support object-oriented programming and, to a limited extent, user-defined language extensions. Coupled with a powerful class library such as that supplied by MacApp, C++ is a sensible platform for serious development where reliability, maintainability, reusability, and efficiency are primary considerations. However, it's also important to consider that C++ is essentially an immature language subject to future growth and mutation. Because C++ was not specifically designed for the Macintosh environment, the integration of language features is not yet seamless, as we've seen.

With the advent of object-oriented programming and higher-level languages like C++ comes the desire of applications programmers to work at as abstract a level as possible, insulated from the often arbitrary details of a particular platform's memory architecture, operating system, or toolbox. With C++ you have the freedom to work closer to the design level, but not without first making an investment in understanding the depths of the language implementation. That is, we think it is *always* important to know what the compiler is doing behind your back. The return on this initial time investment comes in applications that are better designed, more reliable, and easier to upgrade and migrate. This article was written in hopes of reducing some of the pain in this initial time investment, so that you can concentrate on the more enjoyable and productive aspects of developing in C++.

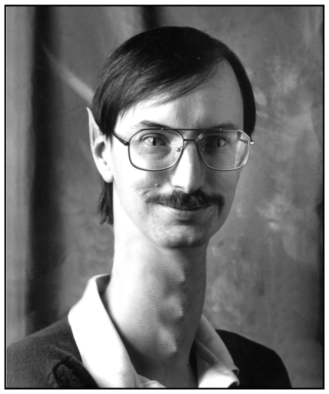
REFERENCES

- Andy Shebanow: "C++ Objects in a Handle-Based World," *develop*, Issue 2, April 1990.
- David Goldsmith and Jack Palevich: "Unofficial C++ Style Guide," *develop*, Issue 2, April 1990.
- Margaret A. Ellis and Bjarne Stroustrup: *The Annotated C++ Reference Manual*, Addison-Wesley, 1990.
- Waldemar Horwat: "The Power of C++," MacHack Conference Proceedings, 1990.
- *MacApp 2.0 Cookbook, Beta Draft*, APDA #M0299LL/C.
- Macintosh Technical Note #88, Signals.
- Macintosh Technical Note #281, Multiple Inheritance and HandleObjects.

Thanks to Our Technical Reviewers
Dave Radcliffe, Larry Rosenstein, Kent Sandvik, Brad Silen•

THE SUBSPACE MANAGER IN SYSTEM 7.0

It has long been a well-kept secret that among the many innovative and controversial aspects of the original Macintosh design—such as bitmapped graphics, iconic interfaces, and excessive disk swapping—was hardware capable of accessing subspace fractal strings. Until now, there has been no system software means of accessing this hardware. But with System 7.0 comes the Subspace Manager, an access path to the underlying subspace transceiver available in every Macintosh. Now the truth can be told—and this article tells it.



HARRY R. CHESLEY

While the Macintosh was being designed, a small group of researchers at the Pacific Alternative Reality Center discovered a simple means of accessing subspace. The method they developed could be incorporated into virtually any integrated circuit, requiring very little space. The Apple development team, aware of this work via personal contacts with the group, decided to incorporate a subspace transceiver into the IWM chip.

Almost from the first day, the decision to include a subspace facility was controversial. Many thought that the device was simply a toy and would keep the Macintosh from being accepted in the business market. Others felt that the device might be dangerous and worried about getting UL approval. Still others were concerned about adding yet another chapter to *Inside Macintosh*.

In the end, a compromise was worked out: the hardware was included in the Macintosh, but no means was provided to access it via the operating system. This allowed the facility to be included in the spec sheet, but kept it from slowing down third-party software developers. An early version of MacWrite used the facility to enable interstellar collaborative editing, but the collaborative aspects were eliminated by Marketing for being too far ahead of their time and thus confusing to users.

When the ASC chip was added to later Macintosh models, an improved subspace transceiver was included. But once again, due to a slight management error—the

HARRY CHESLEY has had a fascination with subspace ever since discovering a wormhole in his bathtub at age five. Spending much of his youth in subspace and his college years as an exchange student on Pluto ("a real Mickey Mouse planet"), Harry has relentlessly pursued the study of subspace, putting him in an ideal position to write this article. Today he lives in subspace with his wife and daughter, and commutes to Apple.•

resources intended to be used to develop the Subspace Manager were instead used for a four-day party in Monterey—no means of accessing the transceiver was made available.

Finally, during the development of System 7.0, someone said, "Hell, we've got everything else in it, why not add the Subspace Manager too." And so it was done. Unfortunately, the contents page of *Inside Macintosh* Volume VI was frozen before a chapter on the Subspace Manager could be added. This article takes the place of that chapter.

ABOUT THE SUBSPACE MANAGER

The Subspace Manager is the part of the operating system that handles communication between the application and the subspace transceiver in the IWM or ASC chip. The Subspace Manager includes routines to access specific subspace dimensional strings and transmission frequencies, subintegral dimensional storage, and the underlying physical constants. Higher-level routines provide access to structured storage as defined by Intragalactic Standards Organization (ISO) document 332.12.2234.2313.22.123a: *Interspecies Data Standards*, Subspace Storage and Retrieval, Structured Formats, Access Methods, subsection J, revision 1822.

SUBSPACE AND FRACTAL STRINGS

To use the Subspace Manager properly, you need to understand what subspace is and how fractal strings work.

The universe is composed of billions (and billions) of strings, all intricately interwoven. Collectively these strings are known as *subspace*. Each individual string is a zero-dimensional point fractally interwoven through the local space-time continuum, bounded by mass concentrations (which distort the spatial geometry and thereby contain the strings). Because a string has zero dimensions, the concept of transit time across the string is meaningless. Because it's fractal, it achieves connectivity with a large area in space.

Each fractal string has a unique fractal number that can be used to identify that specific string. No other string can have exactly the same fractal dimension, as a consequence of Boorman's conservation of dimensionality principle (CoDP—pronounced "cod pee"). CoDP provides a convenient means of addressing a particular string:

```
TYPE CoDPNumber = EXTENDED;
```

THE SUBSPACE TRANSCEIVER

The subspace transceiver in the Macintosh works by creating a subspace resonance chamber, empty of any strings, and then "intruding" a length of a single string. The influence on the string is directly proportional to the length of the string intruded.

Even though the lengths are measured in parsecs, the nature of fractal strings allows the entire length to be intruded into a chamber that fits within a small part of an integrated circuit.

> **Note:** Observant readers will note that although strings are described as zero-dimensional, we can also speak of the length of a portion of string. This is not the contradiction it seems at first glance. However, the mathematics needed to illustrate this fact are beyond the scope of this document.

Subspace string intrusion is commonly called *string sucking* in the technical literature. This is the origin of the popular song “Suckin’ in the Subspace,” written and originally performed by The Fracs. The (totally unsubstantiated) claim is made that during the band’s last performance of this song they hit precisely the right frequency (just under F sharp) and were themselves sucked into subspace using no more instrumentality than a poorly tuned guitar.

> **Warning:** Don’t try this with your Macintosh.

THE INTRAGALACTIC SUBSPACE ENCYCLOPEDIA

In an effort to share information with other species in the same gravitational neighborhood, certain strings have been set aside as a community-access encyclopedia. The format of the encyclopedia is defined by the ISO. Details of these standards are available via the encyclopedia itself. The Subspace Manager includes a high-level interface to the encyclopedia.

> **Warning:** Members of some civilizations have decided not only not to contribute to the encyclopedia, but also to actively disrupt it. These species make changes to existing entries in the encyclopedia, rendering the content questionable and even dangerous. For example, the entry on Earth was changed by one of these species to read “a mostly harmless planet run by small white mice.” Of course, Earth authorities immediately changed it back to “a mostly harmless planet run by large multicolored apes.”

HIGH-LEVEL SUBSPACE MANAGER ROUTINES

This section describes the high-level Pascal interface to the Subspace Manager. Because of space constraints, low-level Pascal routines are not described in this article, but we’re sure you can figure them out yourself.

```
FUNCTION SSInitialize (pi: EXTENDED; e: EXTENDED): OSErr;
```

SSInitialize initializes the hardware, evacuates the intrusion chamber, and tests the local values of pi and e against those given. If pi and e do not match those of the local reality, SSInitialize returns ssWrongReality; otherwise it returns noErr.

TRANSMITTING AND RECEIVING

```
FUNCTION SSTransmit (p: CoDPNumber; VAR count: LONGINT; buffPtr: Ptr):
                OSErr;
```

SSTransmit attempts to send the bytes found at buffPtr, of length count, via the set of strings starting at p. It chooses a series of string numbers based on p, as needed to contain the entire block of data. The algorithm for choosing the string number sequence is defined in the ISO document. Besides being an accepted standard, the sequences are chosen to maximize the string intrusion rate (the "suck"). Upon return, count reflects the actual number of bytes transmitted.

```
FUNCTION SSReceive (p: CoDPNumber; VAR count: LONGINT; buffPtr: Ptr):
                OSErr;
```

SSReceive attempts to receive, from the strings starting at p, the number of bytes specified by count, placing the received bytes in buffPtr. As with SSTransmit, a sequence of string numbers is chosen based on the original p.

ACCESSING THE INTRAGALACTIC ENCYCLOPEDIA

```
FUNCTION SSEncycEntry (p: CoDPNumber; VAR entryTitle: Str255; VAR entry:
                Handle): OSErr;
```

SSEncycEntry attempts to retrieve the encyclopedia entry at string p. It places the title of the entry into entryTitle and returns the contents of the entry in a handle. It is the responsibility of the application to dispose of the entry handle when finished with it.

All entries in the encyclopedia are written in the native language of the originators of the encyclopedia concept—the Herbans. SSEncycEntry automatically translates encyclopedia entries from Herbaneeze into English. However, the system can handle only a limited subset of entries from the entire encyclopedia.

> **Note:** No high-level access is provided to write new entries in the encyclopedia. If, however, you must have write access to the encyclopedia, you can use the low-level interface. This will require you to write an English-to-Herbaneeze translator. System 8.0 will include a bidirectional Herbaneeze translator as part of the Universal Translator Package.

CHANGING REALITY

One of the many consequences of CoDP (the conservation of dimensionality principle) is that each alternative reality has its own unique set of physical constants. In addition, it follows directly from Malanthorpin's theorem of constant universality that only two constants are needed to define a reality, because all other constants can be derived from those two, and that any two constants are sufficient.

The standard reference on subspace is *Subspace Engineering—Theory and Practice* by MacMillon and Boorman (New York: Counterweight Press, 1957). The reader may also be interested in *Hummin' Beings, The Next Stage in Evolution* by Gregor Alman (Chicago: Omega Memes Press, 1997). In this book Alman argues that creatures capable of directly influencing subspace string frequencies are the next logical step in evolution. The fact that he completely misunderstands the concept of evolution and that there are no examples of his so-called hummin' beings listed anywhere in the Intragalactic Encyclopedia doesn't keep the book from being extremely entertaining.•

The Macintosh subspace transceiver has the ability to change the physical constants of the current reality, thus moving the reality into a new Herzhold plane.

```
FUNCTION SSChangeReality (newPi: Extended; newE: Extended): OSErr;
```

SSChangeReality attempts to change the current reality's physical constants. If another reality already exists with the new constants, SSChangeReality returns ssRealityExists. If the reality change was successful, it returns noErr.

Changing the physical constants of reality almost always causes the destruction of all life. To make sure that this is the real intent of the user, a new type of alert is included with the Subspace Manager. The alert is invoked with the function EndOfWorldAlert.

```
FUNCTION EndOfWorldAlert (alertID: INTEGER; filterProc: ProcPtr):
                INTEGER;
```

This alert works the same as the StopAlert, NoteAlert, and CautionAlert functions, except that it uses a different icon, as shown in Figure 1. Physical reality constants should be changed only if the user clicks the OK button in the alert.

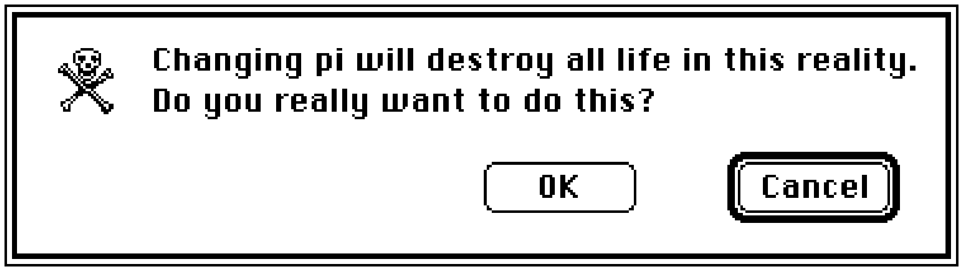

Changing pi will destroy all life in this reality.
Do you really want to do this?

Figure 1
End-of-World Alert

CONCLUSION

This article has described the Subspace Manager available with System 7.0. This new facility provides many powerful capabilities, and should result in many new and exciting applications for the Macintosh.

The story is widely told that sometime in the last century, the legislature of Indiana passed a law declaring that pi would henceforth be exactly 3, with the intent to decrease the cost of teaching that portion of mathematics. This story is untrue: in fact, the law was proposed but was defeated. If the Indiana legislature had only had a Macintosh with a Subspace Manager, they could have actually succeeded in changing pi to 3. And they may yet.

Thanks to Our Technical Reviewers
Harry R. Boorman, Alfred P. MacNewman, Harry R. Malanthorpin •

THE VETERAN NEOPHYTE

IF I HAD A HAMMER . . .

DAVE JOHNSON

The people at ACOT℠ (Apple's Classroom of Tomorrow) recently offered me the opportunity to work on another of their cool projects. I'll tell you all about the project, but—as usual—I'll also veer off into some wild philosophical speculation about computers and programming. So please fasten your seat belts and keep your head and arms inside the magazine at all times.

ACOT was working on a research project in mobile computing: by combining GRiDPAD® computers (notebook MS-DOS machines with pen-based input) and tiny wireless modems, they had created a unit that could be carried around easily in the field and that was continuously connected with other identical units. The software they were developing to run on these machines was a sort of collaborative spreadsheet, so that many separate users could enter and edit data simultaneously, and everyone would be updated continuously. They were going to give these units to kids at an elementary school in Tucson, Arizona, and send 'em out in the desert to collect various kinds of environmental data (temperature, pH, location, number of cacti, and so on). The idea was this: the kids would be able to see not only their own data, but how their data fit into the big picture. Presumably learning is enhanced when a person can see multiple levels of meaning side by side, because the mental "level switching" that has to happen to discern interdependencies can happen faster.

In addition to the spreadsheet, they wanted a user-configurable graphing tool that would enable the user to plot any available value against any other, and to label the data points with a third variable to get a crude sort of a 3-D graph. This is another potentially powerful thing: being able to see the same set of data represented in two different ways side by side should enhance the understanding of the data and how it relates to reality. ACOT was running out of time and needed someone to write the graphing tool. In return for taking on this project I'd get a trip to Tucson to assist in the field trials. It sounded great: I love to program graphics, I have a good friend in Tucson I haven't seen in years, I'd get to learn all about a pen-based computer, and I'd have a chance to participate in some really interesting research.

Unfortunately, this was at a time when my workload, which waxes and wanes over the quarter, was on a steep rise. I had less than ten days to write this graphing tool, and I had to maintain some semblance of responsibility to my regular job. I knew perfectly well that if I took this project it would mean some late night and weekend hacking (something I increasingly try to avoid, at least for code that relates to work), and it would also mean working on an MS-DOS machine, something I had hoped to adroitly sidestep forever. Ah well, what's life without a little adventure? I took it.

Suddenly there I was, sitting in front of this strange machine made by Toshiba, all softly textured gray plastic and glowing plasma orange, surrounded by about thirty pounds of documentation, and the screen says something like "C:\GRID\BIN>." Yikes! What have I done? I'm a Macintosh guy. I can get around OK on UNIX®, thanks to a class I took once, but I've never touched an MS-DOS machine in my life, believe it or not. Now I'm not going to write yet another MS-DOS slam from the Macintosh perspective, but there are two lasting impressions I want to share. First, it took me almost half an hour to copy one directory of files into another the first time I tried, and I'll never forget it. Second, batch files are pretty handy.

DAVE JOHNSON has the best toy collection of anyone around. He says that his favorites either make cool noises, fly, do something surprising, or have just the right number of exclamation points in the product description. His Humming Bee (a rubber band stretched over a cheap wooden frame on the end of a string) hums when you whirl it over your head; Mike Stone's Amazing dip-er-do™ Stunt Plane defies gravity (it's a weighted paper plane that only does tight loops, so no matter which way you throw it, it always comes back to you); and the rattleback—this one's so surprising that Scientific American had to publish something about it (see the article in *Roundabout: The Physics of Rotation in the Everyday World* by Jearl Walker, 1985). But his favorite toy of all is his Macintosh, because it makes cool noises, flies (well, figuratively), and does surprising things all at once. When he's not playing with his (and everyone else's) toys, he enjoys redwoods (wherever he may find them), dogs (his own in particular), clear blue (sky and water), escapist fiction (science and otherwise), and complex mechanical contraptions.•

Mercifully, I didn't have to spend much time in MS-DOS itself. I used Borland's Turbo C® to do the actual development work, and it's a lot like THINK C, my preferred compiler on the Macintosh. Also, the programming interface for the GRiDPAD is very Macintosh-like, so aside from some syntactic differences I felt pretty much at home writing the code. In order to finish on time, though, I did have to go on a rather severe coding binge.

You know that feeling that you get around the sixth or eighth or tenth hour of nonstop digital interaction? Strange tensions, displacement, a weird urgency enclosing every movement, feeling compelled and repulsed simultaneously . . . you've all been there, I'm sure. Isn't it bizarre that computers can create such visceral reactions? Maybe if you do *anything* nonstop for a long time like that it would feel the same, but somehow I don't think so.

Programming is, at least partially, the ability and/or desire to force your mind to be *completely* literal. I have a pet theory that the reason programming is so difficult for many people, and the reason it induces such a strange mind state, is that it's a fundamentally different way of thinking that's not at all natural and must be consciously donned, like a hat that doesn't fit. You know how it sometimes takes a while to get fully into it, and once you're there it takes a while to come out of it? My wife still struggles with that: she doesn't understand that I am in a sort of trance, holding a whole strange world inside my head that is at odds with reality. She'll ask me a simple question, like what should we eat for dinner or have I let the dogs out recently, and it sometimes takes a full ten or fifteen seconds before I can react coherently. And it's not that I'm ignoring her. I just don't have room in my poor overburdened brain for the real world: it's been crowded out by the digital one. And unfortunately, by coming out long enough to answer her, I've lost a lot of ground. It will take another twenty minutes to get back to where I was. I *don't* think, though, that programming has to do that to people forever: it's just that our method of telling computers what to do is still very crude and cumbersome.

How can computers be so . . . I don't know, profound? I mean, they're only machines, right? And they only do one thing really well—they can add—but boy, are they good at it! They can add circles around anything else on the planet. Big circles. And somehow that makes them into what they are: these fluid, configurable, multipurpose tools and toys. You forget, and rightly so, that they're just adding machines on steroids. I once heard Todd Rundgren give a talk at a local SIGGRAPH meeting, and he made the point that computing itself is a poorly understood thing. He compared computers to the handles on tools: if you put a handle on a rock, you've got a hammer. Computers are like handles, but we don't yet know what they're handles to, and I suspect that we won't really know for a long time, if ever. It sure is a lot of fun, though, to grab that handle and start swinging!

Well, I can't tell you the end of the ACOT story, 'cause it hasn't happened yet, but maybe a future column will pick up where this one leaves off. My little graphing tool plugged into the spreadsheet with a minimum of hassle, thankfully, and it seems to be just what they wanted. Next week we get to hand our newly forged handle to the kids and see what they bash into. It might be anticlimactic—maybe they'll just treat it like a fancy pad of paper—but maybe, just maybe, their minds will light up when they grab on.

RECOMMENDED READING

- *Mindstorms: Children, Computers, and Powerful Ideas* by Seymour Papert (Basic Books, Inc., 1980).
- *Good Dog, Carl* by Alexandra Day (The Green Tiger Press, 1985).
- *Shared Minds: The New Technologies of Collaboration* by Michael Schrage (Random House, 1990).

Q *Why would using OpenResFile(fileName) cause a crash when I try to open a Macintosh font file that's already open?*

A The problem stems from the fact that OpenResFile doesn't deal effectively with cases where the resource file is already open. Luckily, there are some relatively new Resource Manager calls that you can and should use instead. They're all documented in the Resource Manager chapter of *Inside Macintosh* Volume VI and in Macintosh Technical Note #214, New Resource Manager Calls.

The call of interest in your case is HOpenResFile. To use it, break down the vRefNum (actually WDRefNum) returned by Standard File into a real vRefNum and dirID by calling PBGetWDInfo, and pass those to HOpenResFile along with the file name. The important part, however, is the permissions byte. If you expect to modify the file, pass fsRdWrPerm in that field. If there's an error of any kind, expect HOpenResFile to return -1, which should serve as a signal that you need to call ResError to find out what went wrong.

Q *My application, which has several units and objects, compiles under MPW 3.1 but not under MPW 3.2. Do forward references to objects work differently with MPW 3.2?*

A MPW 3.2 has a new syntax for forward references to objects. Objects must be declared as externals, as follows:

```
TYPE
   TObjB = OBJECT; external;        { MPW 3.2 requires this }
   TObjA = OBJECT(TObject)
      fFwdRef:   TObjB;

      {methods}
      END;
```

Note that by default the Pascal compiler includes information such as USES in its symbol table resources, so simply using the correct USES and external declarations may not be sufficient; you may find it necessary to invoke the Pascal compiler with the -rebuild option to force it to reconstruct its symbol table resources from scratch.

Q *With the System 7.0 version of the LaserWriter driver, when the user selects Envelope from the Page Setup dialog, the page size returned by the driver is still a standard page: 8.5 x 11. How do you recommend that applications display the page size when the user has chosen a nonstandard page size?*

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 Pete "Luke" Alexander, Mark Baumwell, Jim "Im" Beninghaus, Neil Day, Tim Dierks, Godfrey DiGiorgi, Steve Falkenburg, Bill Guschwan, C. K. Haun, Pete Helme, Dave Hersey, Dennis Hescox, Mark Johnson, Jim Luther, Jim Mensch, Bill Mitchell, Guillermo Ortiz, Greg Robbins, Jack Robson, Kent Sandvik, Gordon Sheridan, Paul Snively, Bryan Stearns, Forrest Tanaka, Vincent Tapia, and Scott "Zz" Zimmerman for the material in this Q & A column. •

A We recommend that you have the page preview show a full page instead of an envelope-sized page. The LaserWriter® driver supports a large number of PostScript® devices, and it can't be sure whether the envelope will be fed on the right, left, or center of the paper tray. If you show the full page, a user can print on any device by putting the text in the correct location for that device.

Manufacturers of PostScript printers can add custom page sizes to the LaserWriter driver. If they do, the representation on the screen will be whatever they decide to define. Applications should not try to interpret custom page sizes. If your application ignores the results returned by the driver, you risk incompatibility down the road.

Q *Why is my Macintosh driver receiving a positive drive number under System 7.0 upon notification of an _Eject call?*

A When the "driver wants a call on eject" bit is set in the flag bytes preceding a drive queue element, _Eject will issue a _Control call with a csCode of 7 to the driver. This _Control call is supposed to inform the driver which disk the OS is attempting to eject, by passing the drive number in the ioVRefNum field of the parameter block.

However, there's a bug in the ROM that only manifests itself when _Eject is given a volume reference number for a disk that has both the "nonejectable" and "driver wants a call on eject" bits set in the drive flag bytes. This bug causes the driver to receive the negative of the drive number, rather than the positive drive number.

The System 7.0 Finder has reversed the order of its calls to _UnmountVol and _Eject, causing it to pass the drive number to _Eject, which then passes it on to the driver correctly. Unfortunately, under previous systems, the Finder passed the volume reference number to _Eject, forcing developers to work around the bug by accepting negative drive numbers; however, a problem could occur now under System 7.0 if positive drive numbers weren't accepted as well.

A number of driver writers have notified us of this problem, but few (so far) have been adversely affected. As it has always been possible for utilities or applications to make _Eject calls with either a volume reference number or a drive number, the proper workaround is to handle both positive and negative drive numbers.

Q *How many active ranges can a Macintosh application have on a shared file? If the answer is more than one, is the limit per application or per machine? If two ranges overlap, are they joined into one range? Can an application nest ranges? For example,*

Have more questions? Need more answers? Take a look at the Dev Tech Answers library on AppleLink (updated weekly) or at the Q & A stack on the *Developer Essentials* disc. •

if an application's user performs an action that forces a record to be locked and later the application locks the full range of the file, does the initial record lock disappear?

A The only way to determine the limit is to hit the limit and get a NoMoreLocks error. The number of range locks supported is a limit of the server platform, and that limit is shared by all users of the server (at least it is with Apple's AppleShare® server software). With Apple's server-based version of AppleShare, approximately 40 locks per user are allowed (for example, if the server allows 25 users, there are 1000 locked ranges total; if the server allows 50 users, there are 2000 locked ranges total; and with File Sharing running under System 7.0, approximately 20 locks are allowed per user). Other vendors may allow more or fewer locked ranges on their implementations of an AppleTalk® Filing Protocol (AFP) server. Notice that the numbers given are per user, not per application. It's assumed that a user probably won't need more than a few locks at a time on a single file.

You cannot have range locks that overlap. You'll get a RangeOverlap error from AFP. All the rules for range locking can be found in the AFP chapter of *Inside AppleTalk* (page 13-56). Additional information on AppleShare limits is available in the Dev Tech Answers library on AppleLink and on the *Developer Essentials* disc.

The February 1991 revision of Macintosh Technical Note #186 covers several important details about PBLockRange and PBUnlockRange that are not in *Inside Macintosh*.

Q *Where can we get our hands on a fix to the ROM bug in the Macintosh IIci and Macintosh IIfx Memory Manager? Word has it that Apple wrote an INIT (MMInit) to fix this problem. Because we're using our application on both the Macintosh IIci and the IIfx, and the application uses many handles, this fix would be appreciated.*

A Under System 6.0.x, Apple had identified a minor problem with the Memory Manager in the Macintosh IIfx, IIci, IIsi, and LC. This problem resulted in a performance degradation in an extremely small number of applications and did not cause system crashes. We believe this was an insignificant problem that affected few applications and very few customers. The problem was not new and was not caused by the introduction of System Software 6.0.5 or 6.0.7.

Based on developer feedback, customer feedback, and extensive in-house testing, Apple has identified very few affected applications. Because the problem affects only an extremely small number of developers, Apple is working with those developers to fix their applications. The best solution, however, is to upgrade to System 7.0, as it includes an enhancement to the Memory Manager to address this issue.

During extensive testing and research, Apple investigated a variety of solutions to enhance the Memory Manager performance. One area researched was a software solution called the Memory Manager INIT (a software "patch"), or MMInit as it's most often called. Through testing we discovered that this patch did not enhance Memory Manager performance and introduced risks such as decreased performance in some mainstream applications.

An unofficial version of the Memory Manager INIT has surfaced. This INIT should not be used, because it has been modified from Apple's experimental version and could cause data corruption, data loss, and crashes. Apple strongly urges that you discard the INIT if you have obtained a copy, and not use it. If any version of this INIT is used under System 7.0, it defeats the enhanced Memory Manager and reintroduces the bugs that were present in the System 6.0.x Memory Manager.

You may still wonder if you have been affected by this problem and how to avoid it under 6.0.x. The problem is most severe when allocating pointers in a heap with a rather large number of handles (on the order of tens of thousands). It's helpful to allocate enough master pointers (via MoreMasters) during initialization. If the Memory Manager has to call MoreMasters later on, not only could it fragment memory, but it could take an exceedingly long time. It's also helpful not to allocate thousands of handles. Besides requiring lots of master pointers, it takes the Memory Manager a long time to crunch through them during heap compaction.

Q *Does setting the is32BitCompatible bit in the 'SIZE' resource have any effect in System 7.0?*

A The alert box that was to be shown for applications with the 'SIZE' resource's is32BitCompatible flag disabled was found to be too confusing for an end user, so the is32BitCompatible flag is not used and the alert box is not displayed in the final System 7.0. (It is, however, displayed in A/UX 2.0 and 2.0.1.) This could change in the future.

Q *Can you describe a procedure for extracting TrueType character outline data directly from the Macintosh system software?*

A Future releases of Macintosh system software probably will include calls for accessing TrueType character outline data. In the meantime, sample code showing how to parse the 'sfnt' outline font resource is available on the *Developer Essentials* disc and on AppleLink in the Developer Support: Developer Services:Developer Technical Support:Developer Essentials:Sample Code folder.

Q *Where can I find information on manipulating TrueType fonts under System 6.0.7?*

A The System 6.0.7 TrueType INIT includes all the outline calls in System 7.0, so everything you need is in *Inside Macintosh* Volume VI. Use Gestalt's gestaltFontMgrAttr selector (described in the "Compatibility Guidelines" chapter of Volume VI, and available in System 6.0.4 and later) to determine whether TrueType is available on the machine in question, and then use outline calls freely and with abandon. The latest MPW release has the header files, called OutlineCalls. Use version 4.1 of Font/DA Mover to move outline fonts under System 6.0.7. Both the TrueType INIT and Font/DA Mover 4.1 are available on AppleLink and on the *Developer Essentials* disc, and you can license Font/DA Mover to include it with your product release by contacting Apple's Software Licensing group.

Q *Why doesn't the StyleWriter® like the SetLineWidth PicComment (182)? It prints the line at a 1-point weight.*

A Most of Apple's QuickDraw printers do not support the SetLineWidth PicComment. Because there's no feedback as to whether the PicComment was successful, use of the comment is problematic. It's not regularly handled, except in PostScript printers.

The best way to make full use of a Macintosh printer's resolution is through the PrGeneral trap. You'll find it thoroughly described in the article "Meet PrGeneral, the Trap That Makes the Most of the Printing Manager" in Issue 3 of *develop*, and also covered in *Inside Macintosh* Volume V. PicComments are discussed in Macintosh Technical Notes #91 and #175.

Q *How can a Macintosh in 24-bit addressing mode read from disk into a GWorld? If the GC card is installed, sometimes the GWorld is a 32-bit one cached on the card. How do we ensure the GWorld will be in main memory?*

A When you create your GWorld, set the keepLocal flag in the flags field of the NewGWorld call. This ensures that the newly created GWorld will be in main memory where you can access it. Read your data into the GWorld and then clear the keepLocal flag with a call to SetPixelState. This will issue a "GWorld has been updated" type of message, causing the GWorld to be cached off to the GC card, giving you the best performance. When you want to load another image, call SetPixelState again, setting the keepLocal flag to bring the GWorld back to main memory. Using this technique you'll be able to load your GWorld and cache it too, and you won't need to recreate the GWorld each time you want to load more data into it.

Q *What are the effects of InitCM, InitCRM, InitCRMUtilities, and similar calls to the Macintosh Communications Tools by nonapplications such as INITs or background processes? What effect do multiple INIT calls have on Communications Tools operations and any existing connections? What do these calls do, and when (and how many times) is it safe to call them?*

A All these initialization calls deal with the process of loading the 'cmtb' resource into the system heap, putting entries into the Communications Toolbox (CTB) dispatch table with references to the current heap zone, and setting up various things having to do with resource management for the operation of the CTB. You can call them over and over again without damaging existing information.

With this new perspective, you can see that a synergy exists between operations of the CTB and the Resource Manager. The CTB needs to be in the appropriate heap zone when a ConnHandle is generated or referenced for it to operate as desired. If you're writing a background process that maintains its own persistent heap zone, you shouldn't have any difficulties using the CTB.

If you're writing an INIT, matters are more complex. You must SetZone(SystemZone) while performing all CTB operations (including initialization, CMGetProcID, and CMNew) to be sure the heap zone will persist after the INIT has completed. Possibly the reason for doing this at INIT time is to create ConnHandles that will persist for CTB access at a later time. It seems reasonable that ConnHandles thus created require a SetZone(SystemZone) by any routine that needs to access them for Read/Write/Status/Delete operations.

Q *Is there some kind of bug with the Macintosh Balloon Help feature in System 7.0? It goes into the application and uses the text from the 'STR#' resource ID 4001 instead of the 'STR#' resource ID 4001 of Finder Help.*

A The System 7.0 Help Manager stores its balloon strings in the Finder™ Help resource file. Under certain circumstances, the Help Manager will access the string resources in your application before it accesses the resources of the Finder Help file. Consequently, problems will occur if your application contains certain 'STR#' and 'STR ' resources.

There are only two circumstances in which your application may be affected. First, if your application uses a 'STR#' resource with ID 4001, the Help Manager will use the first string resource of the list instead of the corresponding resource in Finder Help. When the pointer is placed over your application icon's text on the desktop, the default text, "Change the icon's name by clicking on the name and typing," will be changed to the text stored in 'STR#' 4001 of your application.

Second, if your application has a 'STR ' resource with ID 17251, the Help Manager will use that string resource instead of the corresponding string resource in Finder Help. The default text, "This is an application—a program with which you . . . " will be changed to the text stored in 'STR ' 17251 of your application.

To avoid these problems, you have a few options. You can create your own 'hfdr' resource to override 'STR ' 17251, or you can avoid using 'STR ' 17251 and 'STR#' 4001. If you must use 'STR#' 4001, paste the text "Change the icon's name by clicking on the name and typing" into the first string and use the rest of the strings for your application's use.

Q *How can I determine the intrinsic styles supported by a particular PostScript font?*

A To find the styles that are supported by a particular font, you need to access one of two tables. To find the screen fonts that are available, you can use the font association table, shown on page 38 of *Inside Macintosh* Volume IV. Each entry in the table contains the size and style of the font, as well as the resource ID. Each font in this table should be available for the screen.

When you print a font to a PostScript printer, the Font Manager uses the style-naming table in the family record to create a PostScript name for the font. The table has a list of strings, followed by an entry for each style supported by that particular font. A table for Times might look like this:

Value	**Description**	**Hex Dump**
9	Number of strings <INTEGER>	09
"Times"	Basename of font	05, 54, 69, 6D, 65, 73

6, 7	Suffix index for style 1
	Pascal string that looks like:
	`String[0] := CHR(2);`
	`String[1] := CHR(6);`
	`String[2] := CHR(7);`
		02, 06, 07

6, 8	Suffix index for style 2	02, 06, 08
6, 9	Suffix index for style 3	02, 06, 09
6, 8, 9	Suffix index for style 4	03, 06, 08, 09
"-"	Suffix 1	01, 2D
"Roman"	Suffix 2	05, 52, 6F, 6D, 61, 6E
"Bold"	Suffix 3	04, 42, 6F, 6C, 64
"Italic"	Suffix 4	06, 49, 74, 61, 6C, 69, 63

Except for the first entry, all entries in the table are stored as Pascal strings with a length byte followed by one byte for each character in the string.

The table is used to build the PostScript font name for a particular style. This is a little complicated, so hang on. Let's look at the table again, with each line numbered, and without all the hex:

```
0:  9
1:  "Times"
2:  6, 7
3:  6, 8
4:  6, 9
5:  6, 8, 9
6:  "-"
7:  "Roman"
8:  "Bold"
9:  "Italic"
```

Entry #0 is the number of strings in this table. Entry #1 is the basename of the font. Entries #6-9 are the suffixes that will be appended to the basename for particular styles. Entries #2-5 are the indexes of the suffixes required to create the PostScript names for the different styles (entry 2 = plain, 3 = bold, 4 = italic, and 5= bold+italic).

Let's use entry #2 to build a name:

1. Always start with the basename: "Times"
2. Append the first suffix, #6 in this case: "Times-"
3. Append the second and last suffix, #7: "Times-Roman"

The style-naming table is not documented in the Font Manager chapter of *Inside Macintosh* because it's too specific to PostScript. Instead, it's documented in the *LaserWriter Reference* (Addison-Wesley) on pages 28-35. The manual's description doesn't include examples, so the above example makes things easier to understand.

Q *What is the easiest way to reset a Macintosh color map or palette to the default system color map?*

A Use RestoreDeviceClut(gd:GDHandle). You pass the handle to the device in question, or nil if you want all devices reset. The call is described in the Palette Manager chapter of *Inside Macintosh* Volume VI.

Q *If System 7.0 were the Seven Dwarfs, which dwarf would Virtual Memory be?*

A Sneezy.

APPLE II Q & A

Q *Our Apple IIGS® TextEdit field created with NewControl2 appears to be redrawn within the TESetText call, but Apple IIGS Toolbox Reference Volume 3 says controls won't be redrawn until the next update event. Is this a mistake in the documentation or in our logic?*

A Internally, TextEdit uses control records for all TextEdit records. The main difference between control and noncontrol records is that the control defproc handles many of the standard TextEdit functions without requiring your application to do so. In the case of TESetText, though, TextEdit will always redraw the entire viewRect. This is a mistake in the manual.

Q *We want to load $BC files from a folder when our program is launched. How do we ask the Apple IIGS System Loader to discard these loaded files at application shutdown time? We tried using the main program's master ID from the Memory Manager, but the files are still not unloaded.*

A The Loader is not designed to support more than one program with one user ID. It assumes that InitialLoad or InitialLoad2 will be called with distinct user IDs for each "program" to be controlled individually.

You need to get new user IDs (probably $1000 type) for each module you load, so that you can call UserShutDown on each of them individually when you need to. Don't dispose of the memory at the end; call UserShutDown on each of the IDs and the Loader will take care of the rest. If you're not quitting, you might want to shut them down in zombie state so that they don't have to be reloaded from disk if the memory is available. You can just pass $1000 as the user ID to InitialLoad and it will get a new ID for you and return it on the stack.

Remember that $BC auxiliary types are reserved and must be assigned by Apple Developer Technical Support.

Q *How can I turn off the GS/OS file system cache, or keep it from writing to a disk while my file system optimizer is running?*

A Altering volumes at the block level will confuse GS/OS®, because the ProDOS® File System Translator (FST) keeps copies of file system structures that aren't in the cache. You need to use DWrite, although using DWrite instead of WRITE_BLOCK risks destroying the integrity of any open files on disk, such as the system resource file. If you use WRITE_BLOCK, you must close any open files, including the system resource file if you optimize the boot disk.

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 Matt Deatherage, C. K. Haun, Jim Luther, and Jim Mensch for the material in this Q & A column. •

Have more questions? Need more answers? Take a look at the Dev Tech Answers library on AppleLink (updated weekly) or at the Q & A stack on the *Developer Essentials* disc. •

Once you start optimizing, don't make any calls that could directly or indirectly result in operating system calls—no DA access, no Font Manager calls, no loading tools, nothing. When you're done, GS/OS's internal volume control records (VCRs) will be completely invalid and you'll have to call OSShutDown.

Q *When is it OK to make Apple IIGS system service calls? I'd like to make calls such as MOVE_INFO from a driver that's executing asynchronously.*

A It's OK to make system service calls in response to a GS/OS request, for example. Most of them require the OS environment, such as GS/OS's direct page, but MOVE_INFO, SET_SYS_SPEED, DYN_SLOT_ARBITER, and SIGNAL do not.

When you're not in the GS/OS environment, make sure the proper language card bank of bank 1 is swapped in. Just JSLing there will put you into something that's not a system service call. You can either use the bank $E1 equivalents of MOVE_INFO, SET_SYS_SPEED, and DYN_SLOT_ARBITER, or you can make sure that the right $01 language card bank is enabled:

```
   short
   lda >$E0C068
   pha
   lda >$E0C08B
   lda >$E0C08B
   longmx
; Set up the registers and make your
; JSL  My_Favorite_SysSrv_Call
   short
   lda >$E0C083
   lda >$E0C083
   pla
   sta >$E0C068
   longmx
```

Q *Do I have to write extra program code for my Apple IIGS program to grow a resource?*

A No, it's pretty straightforward. All you have to do to modify the content of any resource (including growing it) is to load the resource in, make any changes you want to the handle (such as change the data inside or call SetHandleSize to make it bigger), and then call MarkResourceChange. The Resource Manager updates the contents of your file when you call UpdateResourceFile. The

Resource Manager recognizes the change in the size of the handle automatically.

Q *The Apple IIGS does not seem to sort out equivalent devices on the Apple Desktop Bus™ (ADB) as the Macintosh does and as outlined in the ADB specification. We want multiple keyboard support, but the Apple IIGS ADB micro just begins reading blindly from addresses 2 and 3, assuming one keyboard and one mouse are attached. Is this information correct?*

A The Apple IIGS does not do the same kind of dynamic device mapping and remapping that the Macintosh ADB Manager does. The "Apple Desktop Bus Tool Set" chapter of the *Apple IIGS Toolbox Reference* gives instructions on how to remap devices dynamically yourself. Essentially, you have two options:

- You can leave the second keyboard address as 2, allowing input from multiple keyboards, but the keyboards' modifier key input will be mixed.
- You can dynamically remap the keyboards in your program or in an INIT (although some forms of remapping require the user to press a key on the device to be remapped) and then a second keyboard will not appear as the standard keyboard—requiring all who use it to do their own ADB requests to get at the information entered from a second keyboard.

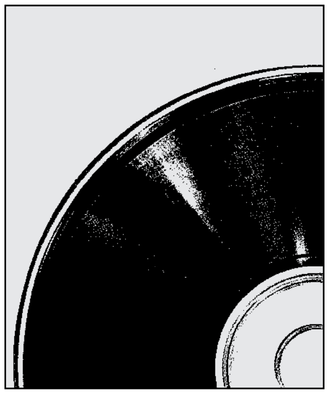
If you don't expect many developers to use a second keyboard, you might just choose to remap it inside any program that uses it. You could write an external library or functions that remap a second keyboard and read from it.

Q *Where can I find documentation on how to recognize SCSI partitions, such as MS DOS partitions, from GS/OS?*

A The documents you'll need are the *GS/OS Reference* (Addison-Wesley) and the *GS/OS Device Driver Reference* (APDA). You can recognize SCSI hard disk partitions programmatically by looking for a SCSI Hard Drive device type ($0005) and a forwardLink or headLink that's nonzero. This will give you all SCSI hard disk partitions, but it won't give you non-SCSI partitions, which have a different device type.

Bit 13 of the Device Characteristics word is for "Linked devices" like partitions, but the *GS/OS Device Driver Reference* says that bit applies to removable media, so not all third-party GS/OS drivers may set that bit for partitions (even though Apple's SCSI hard disk driver does).

Remember that GS/OS requires each partition to appear as a separate device, so there's no support for multiple partitions on one logical device.



YOUR DEVELOPER ESSENTIALS DISC

WHAT'S OLD AND WHAT'S NEW

For each issue of *develop*, there's a corresponding updated version of the *Developer Essentials* CD-ROM disc. If you've subscribed to *develop*, your copy of the disc will be bound into the journal; if you're an Apple Associate or Partner, you'll get your copy in a folder on the *Developer CD Series* disc.

We'll tell you here about some of the headliners in *Developer Essentials*, but you should take some time to browse the disc and see what else you might discover. We'll be adding more as *Developer Essentials* evolves, and we hope you agree that these are tools no developer should be without.

We start out by describing the old standbys (and what they're standing by for) and finish up with descriptions of what's new or improved on this disc.

THE STANDBYS

develop

There's more than one way to browse a magazine (or to look through back issues), and we've given you even more by making *develop* available electronically. With the electronic version of *develop*, you can easily search (by word or with a cumulative index), you can copy the code (or any text that's particularly useful), and you can check out the HyperCard limits we're pushing.

SpInside Macintosh

Of course the most essential documentation for the Macintosh is *Inside Macintosh*, so the *Developer Essentials* disc offers you SpInside Macintosh, an on-line version of Volumes I-V. SpInside Macintosh combines these volumes into a single, searchable electronic form that's cross-referenced with the Macintosh Technical Notes Stack, the Q & A Stack, and the Human Interface Notes Stack.

DTS Technical Notes and Sample Code

All the Apple II and Macintosh Technical Notes and Sample Code programs prepared by Apple's Developer Technical Support group are here for your reference. Technical Notes are updates to existing technical documentation, useful hints and tips, and special coverage of technical topics.

Macintosh Technical Notes Stack

This HyperCard stack incorporates all of the latest Macintosh Technical Notes into a single on-line source, which is cross-referenced with SpInside Macintosh, the Q & A Stack, and the Human Interface Notes Stack.

Q & A Stack

Got a tough development question? The Q & A Stack is a collection of hundreds of the most frequently asked questions answered by the Developer Technical Support group. Organized by subject, this stack includes question and answer pairs as well as cross-references to SpInside Macintosh and the Macintosh Technical Notes Stack.

Human Interface Notes and Stack

These notes will help you develop uniform user interfaces in your Apple II and Macintosh applications. They cover everything from how to use color most effectively (without shortchanging those customers who see everything in black and white) to how to seamlessly incorporate sound.

Apple II
If it's about the Apple II, you'll find it here. We've got documentation on everything from the basic to the most esoteric, as well as MPW IIGS Interfaces, all kinds of disk utilities, system software, and every released version of HyperCard IIGS.

International System Software/HyperCard
The *Developer Essentials* disc includes all the latest international versions of Macintosh system software. In addition, look for the KanjiTalk™ Toolkit, KanjiTalk 6.0 Docs, and the Taiwan Chinese Font Option Kit. (You must have a Macintosh to run DiskCopy and create floppy disks from these images.) The *Developer Essentials* disc also includes the latest international versions of HyperCard in DiskCopy image format.

U.S. System Software/HyperCard
Here you'll find system software versions from 0.1 to 7.0—you can copy them right to a floppy disk using DiskCopy. You'll also find HyperCard U.S. versions 1.2.2, 1.2.5, and 2.0, all of which come complete with an idea stack. Have you ever wondered how many gills there are in a pint? Find the answer in the idea stack.

Programming
No, we won't do it for you, but we'll give you some tools. HyperCard XCMDs (pieces of code used to extend HyperCard functionality), MPW Interfaces & Libraries 3.1, and DefProcs (modules of code for system functionality) are included for your reference.

Snippets
Snippets are small pieces of code that show you one engineer's implementation of something or other. We've tried the snippets to make sure they work, but they haven't benefited from the same testing that *develop* and the rest of our sample code go through. So, before you incorporate a snippet into your code, test it thoroughly and make sure it does what you want it to.

NEW OR IMPROVED

Snippets
Here are the new snippets for this issue.

DemoTextDump 1 This MacApp MPW script contains an example of how to write an MAMake file for C++ that uses load/dump.

DemoTextDump 2 In this two-level dump system, a switch makes it possible to either dump everything or exclude a couple of header files when working with them, which saves time.

DialogBits This sample application shows how to deal with many of the most commonly asked questions about the Dialog Manager.

DoubleBack This sample application shows how to play sounds using double buffering.

ficycle This THINK Pascal™ program lists the files contained in a folder.

FindSysFolder In this code, the FindSysFolder returns the real vRefNum and dirID of the current System Folder. It uses the Folder Manager, if possible, or falls back to SysEnvirons.

FreqForEverChange This sample application shows how to play a sound and how to alter its frequency.

GDevVideo This code shows how to get the parameters out of GDevice records.

InvertedText This sample application gives you cool tricks for printing inverted text.

KeepMeAround This sample code keeps an INIT's resource file open so that code installed by the INIT can access resources stored in the file.

MakeITable This sample application shows how to manipulate a GDevice's inverse table.

Snippets are also available via AppleLink, in the Developer Technical Support folder on the Developer Services Bulletin Board, and in the Dev Tech Answers library.•

MultiHider This sample application, in both MPW and Turbo Pascal® versions, shows how to hide any number of editText fields in a dialog.

MultiPlay This sample application shows how to play a sound several times in a row.

Palette Animation and Palette Animation Gray These sample applications show how to animate the entries of a palette.

PBAllocate This MPW tool shows how to work around a bug in PBAllocate.

SampleSndPlay This sample application shows how to use SndStartFilePlay.

SCSIInquiry This MPW tool shows how to make a SCSI inquiry command to an HD80SC.

SetPDiMC This MPW tool sets the "Printer Driver Is MultiFinder Compatible" flag (see "Learning to Drive" on the *Developer Essentials* disc).

ShowBalloon XFCN This XFCN allows you to use Balloon Help in conjunction with HyperCard. The source code and the demo stack will get you going.

SubLaunch This sample application shows how to launch one application from another.

TestQD and TestVM These sample applications show how to use Gestalt to get information on system features.

UniHider This sample application, in both MPW and Turbo Pascal versions, shows how to hide one editText field in a dialog.

VBLThang.p InstallPersistentVBL takes a VBLRecord and installs it in such a way in this sample code that the VBL will get time even when the application that installs it is switched out.

VertTest This sample application shows how to get information from a 'vers' resource (see Technical Note #189).

Thanks to Guillermo Ortiz for collecting and making sense of these snippets and to Mark Baumwell, Rich Collyer, C. K. Haun, Dave "The Rad" Radcliffe, Kent Sandvik, Paul Snively, and Scott "Zz" Zimmerman for submitting them.•

INDEX

For a cumulative index to all issues of *develop* and a complete source code listing, see the *Developer Essentials* disc.•