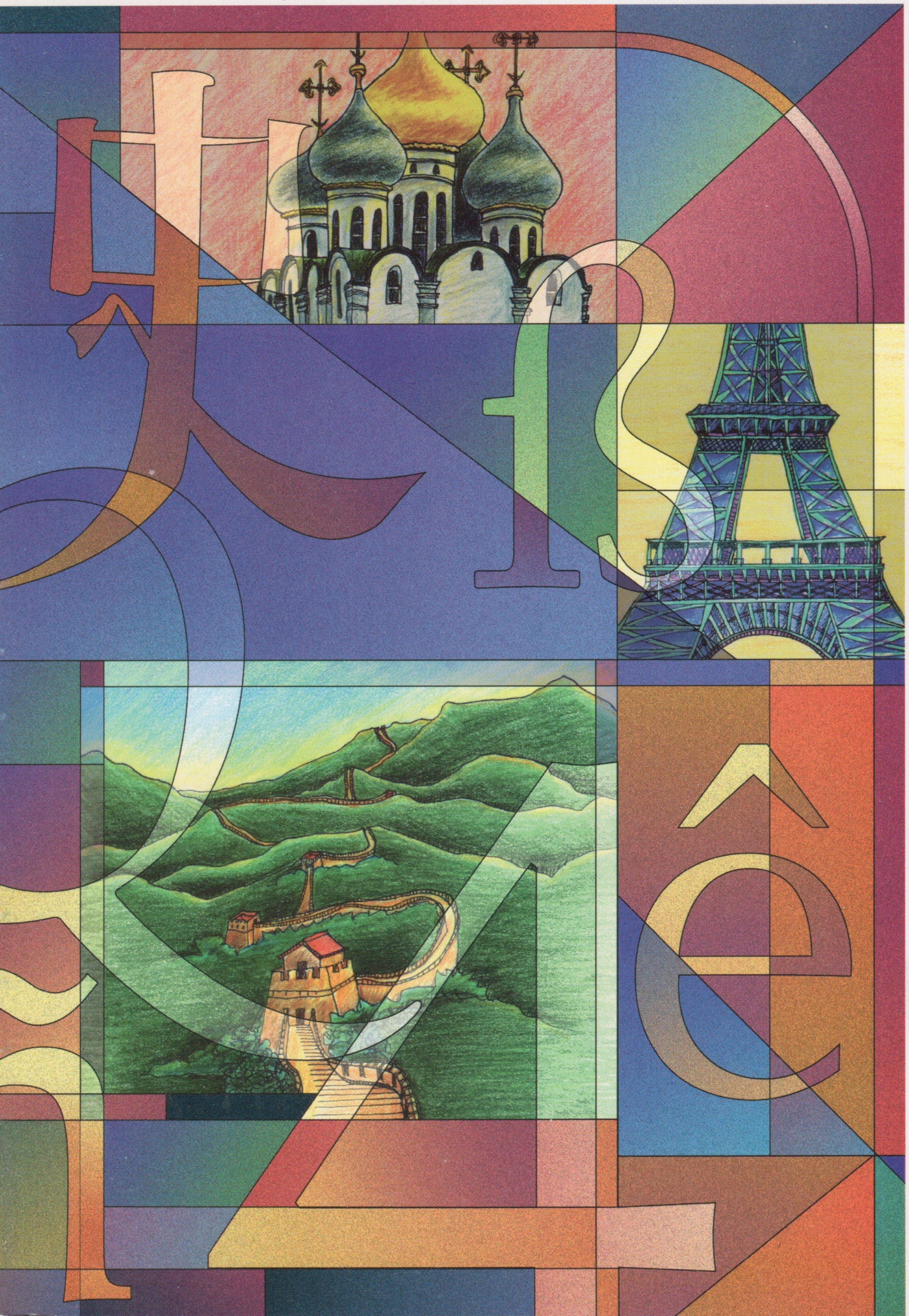
develop

The Apple Technical Journal



Issue 14 June 1993

EDITORIAL STAFF

Editor-in-Cheek *Caroline Rose*

Technical Buckstopper *Dave Johnson*

Our Boss *Greg Joswiak*

His Boss *Dennis Matthews*

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

Managing Editor *Cynthia Jasper*

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

Indexer *Ira Kleinberg*

ART & PRODUCTION

Production Manager *Hartley Lesser*

Art Director *Diane Wilcox*

Technical Illustration *Nurit Arbel, John Ryan*

Formatting *Forbes Mill Press*

Printing *Wolfer Printing Company, Inc.*

Film Preparation *Aptos Post, Inc.*

Production *PrePress Assembly*

Photography *Sharon Beals*

Online Production *Cassi Carpenter*

Maria Mortati of Rucker Huggins created this cover to illustrate the international aspects of writing localizable applications. She used Adobe Photoshop, Adobe Illustrator, and a box of colored pencils.

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

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

CAROLINE ROSE

Dear Readers,

I'd like to talk about a change that you've no doubt already noticed: printed *develop* is no longer accompanied by the *Developer CD Series* disc, but instead comes with its own CD as in days of old. Yes, we've come full circle: *develop* had its own CD for Issues 1 through 7; then, starting with Issue 8, it was accompanied by the *Developer CD Series* disc, which offered a superset of the contents of *develop*'s own CD. The Developer CD is actually a monthly CD that's mailed to members of Apple's developer programs and to subscribers to the Apple Developer Mailing (formerly the APDA Technical Information Mailing). At the time, it seemed simpler and more beneficial all around to just take every third Developer CD and make it the CD that accompanies *develop*.

So why did we change back? Well, it became increasingly common for Developer CD contributors to submit valuable software (or documentation) that was considered appropriate for the "monthly mailing" but *not* for develop. The bargain low price of *develop* simply wasn't adequate to cover the value of the software. (An example that some of you noticed is WorldScript, which made it onto the monthly Developer CD but was pulled at the last minute from the Developer CD accompanying *develop*; see the Letters section.) Keeping valuable materials off the Developer CD or raising the price of a *develop* subscription would have been far less desirable than the option finally chosen, to restore *develop* to having its own disc that's a subset of the Developer CD. This new disc, which comes tucked into *develop* like a bookmark, is called the *develop Bookmark* CD.

This is not to say that *develop* isn't still a bargain! Even without the CD, it's a high-quality journal with articles that have passed rigorous review by engineers at Apple and are also enjoyable to read. Throw in a CD containing the code described in the articles, and already you're getting your money's worth (IMHO). But *develop*'s CD gives you even more than that: it contains all back issues and their code, Tech Notes, Sample Code from the Developer Support Center, *Apple Direct*, the Apple DocViewer application for electronic browsing, and assorted useful tools and documentation.

On the *develop Bookmark* CD, you'll notice some things missing from past CDs you received with *develop*, primarily system software and *Inside Macintosh* (both old and new). If you still want to get these, consider subscribing to the Apple Developer

CAROLINE ROSE (AppleLink CROSE) was a programmer for seven years, until around the time "goto" went out of fashion. In those days, C was just the third letter of the alphabet and OOPS a loud interjection. Caroline's technical background came in handy when she was hired by Apple to work on *Inside Macintosh* Volumes I–III (may they rest in peace). What she did next after that turned out to be merely an interruption in her career at Apple, where she returned two years ago to edit *develop*. Speaking of "Oops," Caroline betrayed both her Italian heritage and her intense perfectionism by misspelling "mozzarella" in her bio in Issue 13; she received as punishment 30 lashes with a wet strand of spaghetti. Her next vacation will be spent on the Isle of Spice (trivia question: where?), and she is already brushing up on how to spell "callaloo" and "ylang-ylang."•

Mailing so that you'll once again receive the monthly *Developer CD Series* disc, which now includes more than ever before. And keep an eye out for other products in the APDA catalog that may include what you want. (See the inside back cover for information about APDA.)

For those *develop* subscribers who receive the Developer CD monthly and don't care which disc *develop* comes with, you'll continue to have a beautiful printed copy of the journal that you can curl up with and easily pass along (think of it as "*develop* unplugged"). The months that *develop* is published, the Technical Documentation edition of the Developer CD will be included in the monthly mailing. (See the March issue of *Apple Direct* for details on the new three-edition *Developer CD Series*.)

To clean the CD slate, so to speak, printed back issues of *develop* will now be accompanied by the Bookmark CD corresponding to this issue, rather than the Developer CDs they were originally paired with. Remember, all *develop* code is kept up to date on the CD, and bugs fixed as necessary, so the latest CD is always the best one to refer to for code.

Since *develop* readers will have either the *develop Bookmark* CD or the *Developer CD Series* disc, how does *develop* now refer to the disc containing the code it describes? After endless debate on this crucial issue, we decided on "this issue's CD." Clever, huh? We're proud of it.

Speaking of pride, I can't help but mention that *develop* won the Excellence award in the Society for Technical Communication's 1992 Northern California Technical Publications and Arts Competition. Please don't forget to let us know how we can make *develop* even better!

CRose

Caroline Rose
Editor

SUBSCRIPTION INFORMATION
Subscriptions to *develop* are available through APDA (see inside back cover for APDA information), or you can use the subscription card in the back of this issue. Please address all subscription-related inquiries to *develop,* Apple Computer, Inc., P.O. Box 531, Mt. Morris, IL 61054 (or AppleLink DEV.SUBS).•

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

LETTERS

DEBUGGING LESSON

I just finished reading the debugging article in Issue 13 and I wondered if you had ever written up anything about this one:

```
TYPE
  LHdl = ^LPtr;
  LPtr = ^LONGINT;

{$S Fill_Seg}
PROCEDURE FillWithData(h:Handle);
  BEGIN
   h^^ := 23;
  END;

{$S Main}

PROCEDURE CanYouSpotTheProblem;

  VAR
   h :LHdl;

  BEGIN
   h := LHdl(NewHandleClear(Size));
   FillWithData(h);
  END.
```

I call this "Routines That Don't Move Memory, Most of the Time." The problem comes about because of the hidden trap to LoadSeg. If all the code resources are preloaded, everything is OK. But if they aren't, the call to FillWithData may cause a memory move, and **h** is not locked. This is especially vile and nasty when, like us, you run in a very limited heap and unload your code frequently.

I know that if you read *Inside Macintosh* Volume II very carefully this is clear. But most of the interesting examples are the ones that are well documented but hard to catch.

—Bob Tipton

Thanks for the example. You're right, of course; calling routines in other, unloaded segments can indeed move memory. This is documented in various places, but unless you understand very clearly how your program occupies memory and how the Segment Loader works, it's easy to overlook. Judicious use of a good heap scramble/purge will catch this one before it catches you.

Thanks for writing!

—Dave Johnson

WHERE IN THE WORLD(SCRIPT)?

I was looking for the WorldScript folder on Issue 12's Developer CD, "Wayne's GWorld" (Dev.CD Nov/Dec 92). According to the Contents Catalog stack, the WorldScript folder should be found in this path: Dev.CD Nov/Dec 92: System Software: WorldScript. But it's not there. I talked with a friend of mine who has the same CD, and he found the folder where it's supposed to be. We checked our CDs and found out that his says 564.6 MB on disc, 69.7 MB available, while mine says 555.5 MB on disc, 78.9 MB available. What's going on?

—Toru Kawate

P.S. Thank you for the fine journal and CD. I really enjoy them.

I received Issue 12 of *develop* with the Nov/Dec 1992 *Developer CD Series* disc. I enjoy the Developer CD but had the following trouble: I used the Contents Catalog on the CD and found the title "WorldScript." But I didn't find the folder at the pathname Dev.CD Nov/Dec 92: System Software: WorldScript.

—Hirokazu Yaguchi

WE WORRY WHEN YOU DON'T WRITE
We welcome timely letters to the editors, especially from readers reacting to articles that we publish in *develop*. Letters should be addressed to Caroline Rose (or, if technical *develop*-related questions, to Dave Johnson) at Apple Computer, Inc., 20525 Mariani Avenue, M/S 75-2B, Cupertino, CA 95014 (AppleLink CROSE or JOHNSON.DK). All letters should include your name and company name as well as your address and phone number. Letters may be excerpted or edited for clarity (or to make them say what we wish they did).•

At the last minute, WorldScript was pulled from the CD accompanying develop, *but there wasn't enough time to remove it from the Contents Catalog. WorldScript did, however, remain on the Developer CD Series disc received monthly by members of Apple's developer programs.*

The reason for this is tied in with why develop *now has its own CD, separate from the Developer CD Series disc; for details, see the editorial on page 2.*

—Caroline Rose

WHEN TO EXPECT DEVELOP

Sorry to disturb you, but because I've had problems with my subscription (two copies, no copies) — which I've worked out, I think, with the DEV.SUBS people — I'm wondering: Is there some sort of publication schedule available that will allow me to fret about this without disturbing anyone? ("Gee, it's two weeks past when this should be out . . . what am I missing out on?!")

Y'all are doing a fine job. Just be sure to run your issues through a Quark SmugCheck to keep the content only kinda wacky and smug without being too much that way.

Thanks!

—J. C. Burns

I'm sorry you've had subscription problems, but I'm grateful that you pursued them with DEV.SUBS rather than sending an AppleLink to CROSE or DEVELOP; many developers make the mistake of contacting develop *staff with these problems, but it's really out of our realm. We are, however, doing our best to make the problems go away.*

If you're located in the U.S., you should normally receive develop *around the first of March, June, September, and December. (If you're outside the U.S., it's hard to say, because* develop *might need to wait for other materials to be merged into the same mailing.)*

Thanks for the tip on SmugCheck; it's just what we need!

—Caroline Rose

M.TN.DESIGNATIONS

I notice you still add the numbers in references to Tech Notes. If dropping the numbers were such progress, that wouldn't be necessary. *And* the new alphanumeric reference codes are not used.

I'm a technical person who is used to abbreviations everywhere for conciseness. Besides, citing with full titles adds noise and a lot of repetitiveness.

—Peter Fink

We decided to include the old numbers in references to Tech Notes to help out those people who like to use hard copy but still have the numbered versions. We'll stop doing that once enough time has elapsed that we expect everyone will have the new, unnumbered versions.

For a variety of reasons, we decided not to include the new designations, such as "M.PT.StandAloneCode," which identify the category/folder for each Note. We figured people could find the Notes in the new file organization easily enough — a Note on QuickTime is in the QuickTime folder, right? But we've since realized that the category isn't always obvious. So *starting with this issue we refer to Notes by category — for example, "See the Macintosh (Platforms & Tools) Technical Note, 'Stand-Alone Code, ad nauseam.'"*

Many developers simply look up specific notes in the various Tech Note indexes, and don't use the special designations at all. For those who do use them, they've been improved to correlate more closely to the Notes' titles; for more on this and other recent Tech Note improvements, see the box on this page.

We're always open to suggestion, but so far you're the only one to mention this. If others reading this reply have similar feedback, I hope they'll let us know.

—Caroline Rose

TECH NOTES AND Q&AS: STATE OF THE UNION

BY NEIL DAY, TECH NOTE AND SAMPLE CODE POOH-BAH

On the August 1992 *Developer CD Series* disc, the new organization of Macintosh Technical Notes made its debut. Since then, we've been listening for suggestions for improvement. The vast majority of the feedback has been positive, but you did point out a few areas for attention:

- Better name correlation: Filenames, titles, and designations like "M.PT.StandAloneCode" needed to be more tightly bound to one another. The latest release fixes this problem; Tech Notes should be easier to find across the print and electronic versions.
- Organization: Tech Notes are now organized alphabetically by title within each section. This was always the intention, but because filenames and titles didn't really match, things were a little haphazard.
- Locating items in print: We've added the designation to the footer of each note, so you can quickly tell where you are as you flip through the pages.

Many of you have asked how to quickly locate the most recently written and updated Tech Notes. On the CD, aliases to the latest Notes can always be found in the "What's new on this CD?" folder. Using the Finder's View by Date option on the category folders is a handy way to see the most recent updates.

Also, please note that to group related information more logically, we've integrated Q&As into the Tech Note library. Q&As appear at the beginning of every section in the print version, and have the label "Essential" in the electronic version (or a different label if you've changed that label name on your system).

Many of you noticed that the Tech Note and Q&A stacks have gone away: the Tech Note library is now available in Apple DocViewer format (as well as Microsoft Word files). You should find the searching and viewing in Apple DocViewer much more usable; please check it out.

These changes should make information much easier to find. As always, if you have suggestions or ideas for improving the Tech Note library, please let me know!

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

WRITING LOCALIZABLE APPLICATIONS

More and more software companies are finding rich new markets overseas. Unfortunately, many of these developers have also discovered that localizing an application involves a lot more than translating a bunch of STR# resources. In fact, localization often becomes an unexpectedly long, complex, and expensive development cycle. This article describes some common problems and gives proactive engineering advice you can use during initial U.S. development to speed your localization efforts later on.

JOSEPH TERNASKY AND BRYAN K. ("BEAKER") RESSLER

Most software localization headaches are associated with text drawing and character handling, so that's what this article stresses. Four common areas of difficulty are:

- keyboard input (specifically for two-byte scripts)
- choice of fonts and sizes for screen display
- date, time, number, and currency formats and sorting order
- character encodings

We discuss each of these potential pitfalls in detail and provide data structures and example code.

PRELIMINARIES

Throughout the discussion, we assume you're developing primarily for the U.S. market, but you're planning to publish internationally eventually (or at least you're trying to keep your options open). As you're developing your strategy, here are a few points to keep in mind:

- Don't dismiss any markets out of hand — investigate the potential rewards for entry into a particular market and the features required for that market.

JOSEPH TERNASKY wrote accounting software for a "Big Eight" firm until a senior partner recruited him into the Order of the Free Masons. He showed great promise as an Adept, and the Order sent him to the Continent to continue his studies under the notorious Aleister Crowley, founder of the Temple of the Golden Dawn. After years of study in the Great Art, Joseph was sent back to America to accelerate the breakdown of civil order and the Immanentizing of the Eschaton. He resumed his former identity and now spends his remaining years adding hopelessly complicated international features to the Macintosh system software and various third-party applications.•

- The amount of effort required to support western Europe is relatively small. Depending on the type of application you're developing, the additional effort required for other countries isn't that much more. There's also a growing market for non-Roman script systems inside the U.S.
- The labor required to build a *truly global* program is much less if you do the work up front, rather than writing quick-and-dirty code for the U.S. and having to rewrite it later.
- Consider market growth trends. A market that's small now may be big later.

This article concentrates on features for western Europe and Japan because those are the markets we're most familiar with. We encourage you to investigate other markets on your own.

LINGO LESSON 101

This international software thing is rife with specialized lingo. For a complete explanation of all the terms, see the hefty "Worldwide Software Overview," Chapter 14 of *Inside Macintosh* Volume VI. But we're not here to intimidate, so let's go over a few basic terms.

Script. A writing system that can be used to represent one or more human languages. For example, the Roman script is used to represent English, Spanish, Hungarian, and so on. Scripts fall into several categories, as described in the next section, "Script Categories."

Script code. An integer that identifies a script on the Macintosh.

Encoding. A mapping between characters and integers. Each character in the character set is assigned a unique integer, called its *character code*. If a character appears in more than one character set it may have more than one encoding, a situation discussed later in the section "Dealing With Character Encodings." Since each script has a unique encoding, sometimes the terms *script* and *encoding* are used interchangeably.

Character code. An integer that's associated with a given character in a script.

Glyph. The displayed form of a character. The glyph for a given character code may not always be the same — in some scripts the codes of the surrounding characters provide a context for choosing a particular glyph.

Line orientation. The overall direction of text flow within a line. For instance, English has left-to-right line orientation, while Japanese can use either top-to-bottom (vertical) or left-to-right (horizontal) line orientation.

BRYAN K. ("BEAKER") RESSLER (AppleLink ADOBE.BEAKER) had his arm twisted by *develop* editor Caroline Rose, forcing him to write *develop* articles on demand. He resides in a snow cave in Tibet, where he fields questions ranging from "Master, what is the meaning of life?" to "Master, why would anyone want to live in a Tibetan snow cave and answer questions for free?" When he's not busy answering the queries of his itinerant clientele, he can usually be found writing some esoteric sound or MIDI application. Back in his days of worldly endeavor, Beaker wrote some of the tools that were used for testing Kanji TrueType fonts, and then worked on System 7 in the TrueType group. Hence the retreat to his current colder but more enlightened and sane environment.•

Character orientation. The relationship between a character's baseline and the line orientation. When the line orientation and the character baselines go in the same direction, it's called *with-stream* character orientation. When the line orientation differs from the character baseline direction, it's called *cross-stream* character orientation. For instance, in Japanese, when the line orientation is left-to-right, characters are also oriented left-to-right (with-stream). Japanese can also be formatted with a top-to-bottom (vertical) line orientation, in which case character baselines can be left-to-right (cross-stream) or top-to-bottom (with-stream). See Figure 1.

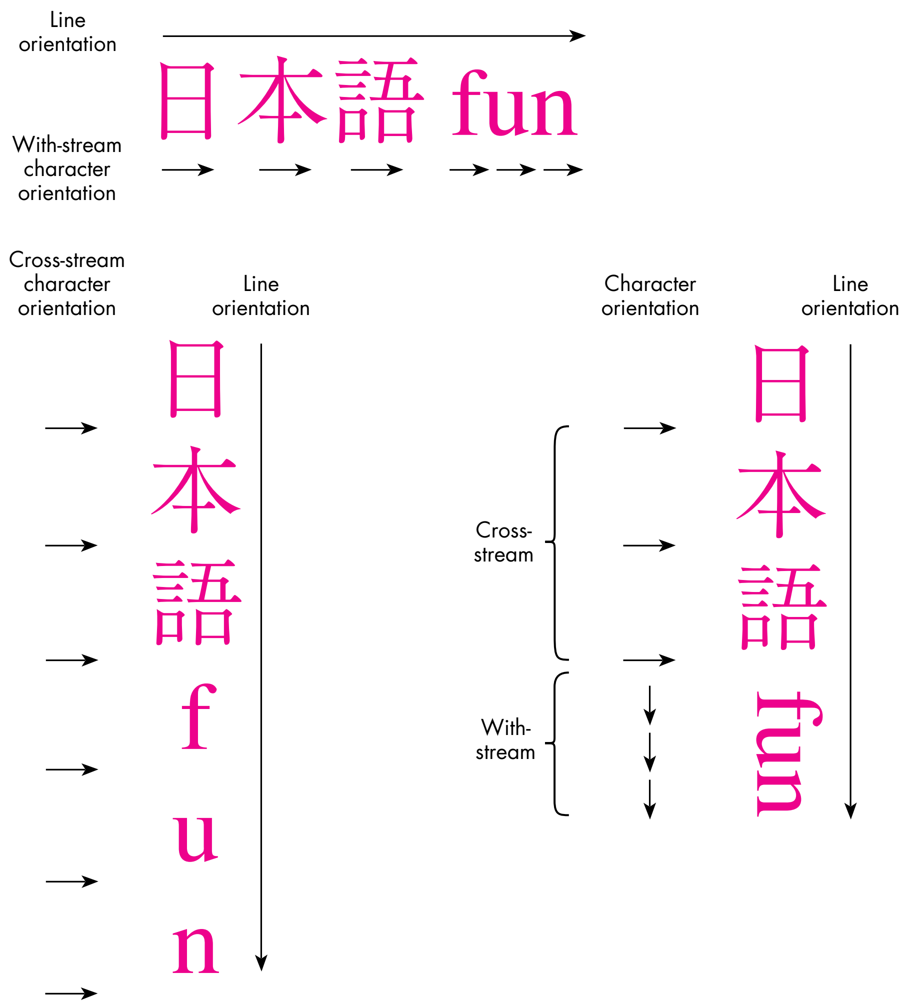


Figure 1
Line and Character Orientation in Mixed Japanese/English Text

SCRIPT CATEGORIES

Scripts fall into different categories that require different software solutions. Here are the basic categories:

- *Simple scripts* have small character sets (fewer than 256 characters), and no context information is required to choose a glyph for a given character code. They have left-to-right lines and top-to-bottom pages. Simple scripts encompass the languages of the U.S. and Europe, as well as many other countries worldwide. For example, some simple scripts are Roman, Cyrillic, and Greek.
- *Two-byte scripts* have large character sets (up to 28,000 characters) and require no context information for glyph choice. They use various combinations of left-to-right or top-to-bottom lines and top-to-bottom or right-to-left pages. Two-byte scripts include the languages of Japan, China, Hong Kong, Taiwan, and Korea.
- *Context-sensitive scripts* have a small character set (fewer than 256 characters) but may have a larger glyph set, since there are potentially several graphic representations for any given character code. The mapping from a given character code to a glyph depends on surrounding characters. Most languages that use a context-sensitive script have left-to-right lines and top-to-bottom pages, such as Devanagari and Bengali.
- *Bidirectional scripts* can have runs of left-to-right and right-to-left characters appearing simultaneously in a single line of text. These scripts have small character sets (fewer than 256 characters) and require no context information for glyph choice. Bidirectional scripts are used for languages such as Hebrew that have both left-to-right and right-to-left characters, with top-to-bottom pages.

There are a few exceptional scripts that fall into more than one of these categories, such as Arabic and Urdu. Arabic, for instance, is both context sensitive and bidirectional.

Now with the preliminaries out of the way, we're ready to discuss some localization pitfalls.

KEYBOARD INPUT

Sooner or later, your users are going to start typing. You can't stop them. So *now* what do you do? One approach is to simply ignore keyboard input. While perfectly acceptable to open-minded engineers like yourself, your Marketing colleagues may find this approach unacceptable. So, let's examine what happens when two-byte script users type on their keyboards.

Obviously, a Macintosh keyboard doesn't have enough keys to allow users of two-byte script systems to simply press the key corresponding to the one character they want out of 28,000. Instead, two-byte systems are equipped with a software *input method*, also called a *front-end processor* or *FEP*, which allows users to type phonetically on a keyboard similar to the standard U.S. keyboard. (Some input methods use strokes or codes instead of phonetics, but the mechanism is the same.)

As soon as the user begins typing, a small *input window* appears at the bottom of the screen. When the user signals the input method, it displays various *readings* that correspond to the typed input. These readings may include one or more two-byte characters. There may be more than one valid reading of a given "clause" of input, in which case the user must choose the appropriate reading.

When satisfied, the user accepts the readings, which are then flushed from the input window and sent to the application as key-down events. Since the Macintosh was never really designed for two-byte characters, a two-byte character is sent to the application as two separate one-byte key-down events. Interspersed in the stream of key-down events there may also be one-byte characters, encoded as ASCII.

Before getting overwhelmed by all this, consider two important points. First, *the input method is taking the keystrokes for you.* The keystrokes the user types are not being sent directly into your application — they're being processed first. Also, since the user can type a lot into the input method before accepting the processed input, you can get a big chunk of key-down events at once.

So let's see what your main event loop should look like in its simplest form if you want to properly accept mixed one- and two-byte characters:

```
// Globals
unsigned short gCharBuf;      // Buffer that holds our (possibly two-byte)
                              // character
Boolean        gNeed2ndByte;  // Flag that tells us we're waiting for the
                              // second byte of a two-byte character

void EventLoop(void)
{
   EventRecord    event;         // The current event
   short          cbResult;      // The result of our CharByte call
   unsigned char  oneByte;       // Single byte extracted from event
   Boolean        processChar;   // Whether we should send our application
                                 // a key message

   if (WaitNextEvent(everyEvent, &event, SleepTime(), nil)) {
      switch (event.what) {
         . . .
```

System 7.1 provides a standard inline input interface and the system input method supports inline input. With inline input, the input translation process can occur within the document window, and no input window is used.•

```
         case keyDown:
         case autoKey:
            . . .
            // Your code checks for Command-key equivalents here.
            . . .
            processChar = false;
            oneByte = (event.message & charCodeMask);
            if (gNeed2ndByte) {
               // We're expecting the second byte of a two-byte character.
               // So OR the byte into the low byte of our accumulated
               // two-byte character.
               gCharBuf = (gCharBuf << 8) | oneByte;
               cbResult = CharByte((Ptr)&gCharBuf, 1);
               if (cbResult == smLastByte)
                  processChar = true;
               gNeed2ndByte = false;
            } else {
               // We're not expecting anything in particular. We
               // might get a one-byte character, or we might get the
               // first byte of a two-byte character.
               gCharBuf = oneByte;
               cbResult = CharByte((Ptr)&gCharBuf, 1);
               if (cbResult == smFirstByte)
                  gNeed2ndByte = true;
               else if (cbResult == smSingleByte)
                  processChar = true;
            }

            // Now possibly send the typed character to the rest of the
            // application.
            if (processChar)
               AppKey(gCharBuf);
            break;

         case . . .
      }
   }
}
```

CharByte returns smSingleByte, smFirstByte, or smLastByte. You use this information to determine what to do with a given key event. Notice that the AppKey routine takes an unsigned short as a parameter. That's very important. For an application to be two-byte script compatible, you need to *always* pass unsigned shorts around for a single character. This example is also completely *one-byte* compatible — if you put this event loop in your application, it works in the U.S.

The example assumes that the grafPort is set to the document window and the port's font is set correctly, which is important because the Script Manager's behavior is governed by the font of the current grafPort (see "Script Manager Caveats"). Although this event loop works fine on both one-byte and two-byte systems, it could be made more efficient. For example, since input methods sometimes send you a whole mess of characters at a time, you could buffer up the characters into a string and send them wholesale to AppKey, making it possible for your application to do less redrawing on the screen.

AVOIDING FONT TYRANNY

Have you ever written the following lines of code?

```
void DrawMessage(short messageNum)
{
   Str255   theString;

   GetIndString(theString, kMessageStrList, messageNum);
   TextFont(geneva);
   TextSize(9);
   MoveTo(kMessageXPos, kMessageYPos);
   DrawString(theString);
}
```

If so, you're overdue for a good spanking. While we're very proud of you for putting that string into a resource like a good international programmer, the font, size, and pen position are a little too, well, specific. Granted, it's hard to talk yourself out of using all those nice constants defined in Fonts.h, but if you're trying to write a localizable application, this is definitely the *wrong* approach.

SCRIPT MANAGER CAVEATS

When you use a char to store a character or part of a character, use an unsigned char. In two-byte scripts, the high byte of a two-byte character often has the high bit set, which would make a signed char negative, possibly ruining your day. The same goes for the use of a short to store a full one- or two-byte character — use an unsigned short.

Another important point is that most Script Manager routines rely on the font of the current grafPort for their operation. That means you should *always* be sure that the port is set appropriately and that the font of the current port is correct before making any Script Manager calls.

A new set of interfaces has been provided for System 7.1. While the old Script Manager's text routines still work, the new routines add flexibility. For example, you can use CharacterByteType instead of CharByte.

A better approach is to do this:

```
TextFont(applFont);
TextSize(0);
GetFontInfo(&fontInfo);
MoveTo(kMessageXPos, kMessageYMargin + fontInfo.ascent +
   fontInfo.leading);
```

Since applFont is always a font in the system script, and TextSize(0) gives a size appropriate to the system script, you get the right output. Plus, you're now positioning the pen based on the font, instead of using absolute coordinates. This is important. For instance, on a Japanese system TextSize(0) results in a point size of 12, so the code in the preceding example might not work if the pen-positioning constants were set up to assume a 9-point font height.

If you want to make life even easier for your localizers, you could eliminate the pen-positioning constants altogether. Instead, use an existing resource type (the 'DITL' type is appropriate for this example) to store the layout of the text items in the window. Even though you're drawing the items yourself, you can still use the information in the resource to determine the layout, and the localizers can then change the layout using a resource editor — which is a lot better than hacking your code.

There are some other interesting ways to approach this problem. Depending on what you're drawing, the Script Manager may be able to tell you both which font and which size to use. Suppose you need to draw some help text. You can use the following code:

```
void DrawHelpText(Str255 helpText, Rect *helpZone)
{
   long     fondSize;

   fondSize = GetScript(smSystemScript, smScriptHelpFondSize);
   TextFont(HiWord(fondSize));
   TextSize(LoWord(fondSize));
   NeoTextBox(&helpText[1], helpText[0], helpZone, GetSysJust(),
      0, nil, nil);
}
```

Here the Script Manager tells you the appropriate font and size for help text. On a U.S. system, that would be Geneva 9; on a Japanese system, it's Osaka 9. NeoTextBox is a fast, flexible replacement for the Toolbox routine TextBox and is Script Manager compatible. You can learn more about NeoTextBox by reading "The TextBox You've Always Wanted" in *develop* Issue 9.

The Script Manager has some other nice combinations:

```
smScriptMonoFondSize    // Default monospace font and size (use when
                        // you feel the urge to use Courier 12)
smScriptSmallFondSize   // Default small font and size (use when you
                        // feel the urge to use Geneva 9)
smScriptSysFondSize     // Default system font and size (use when you
                        // feel the urge to use Chicago 12)
smScriptAppFondSize     // Default application font and size (use as
                        // default document font)
```

The various FondSize constants are available only in System 7. If you're writing for earlier systems, you should *at least* use GetSysFont, GetAppFont, and GetDefFontSize, as described in Chapter 17 of *Inside Macintosh* Volume V. And if you're too lazy to do even that, *please* use TextFont(0) and TextSize(0) to get the system font, which will be appropriate for the system script. This is, by the way, how grafPorts are initialized by QuickDraw. In other words, if you don't touch the port, it will already be set up correctly for drawing text in the system script.

INTERNATIONAL DATING

Before you get too excited, you should know that we're not talking about the true-love variety of date here. No, we're talking about something much more tedious — input and output of international dates, times, numbers, and currency values. First we'll look at output formatting, and then input parsing.

OUTPUT OF DATES, TIMES, NUMBERS, AND CURRENCY VALUES

To output dates, times, numbers, and currency values (which we'll call *formatted values*), you need to know the script you're formatting for. This can be a user preference, or you can determine the script from the current font of the field associated with the value you're formatting (use Font2Script).

You can use these International Utilities routines to format dates, times, and numbers:

- Use IUDateString for formatting a date.
- Use IUTimeString for formatting a time.
- Use NumToString for simple numbers without separators.
- Use Str2Format and FormatX2Str for complete number formatting with separators.

Formatting a currency value is a bit trickier. You have to format the number and then add the currency symbol in the right place. We'll show you how to get the currency symbol and the positioning information from the 'itl0' resource.

First, let's look at an example of date and time formatting:

```
#define kWantSeconds    true     // For IUTimeString
#define kNoSeconds      false

unsigned long     secs;
Str255            theDate, theTime;

// Get the current date and time into Pascal strings.
GetDateTime(&secs);
IUDateString(secs, shortDate, theDate);
IUTimeString(secs, kNoSeconds, theTime);
```

Formatting a number with FormatX2Str is a little more complicated, because FormatX2Str requires a canonical number format string (type NumFormatString) that describes the output format. You make a NumFormatString by converting a literal string, like

```
##,###.00;-##,###.00;0.00
```

The strings are in the format

```
positiveFormat;negativeFormat;zeroFormat
```

where the last two parts are optional. The example string would format the number 32767 as 32,767.00, -32767 as -32,767.00, and zero as 0.00. The exact format of these strings can be quite complicated and is described in *Macintosh Worldwide Development: Guide to System Software.*

The following handy routine formats a number using a format read from a string list. You provide the string list resource and specify which item in the list to use when formatting a given number.

```
OSErr FormatANum(short theFormat, extended theNum, Str255 theString)
{
   NItl4Handle       itl4;
   OSErr             err;
   NumberParts       numberParts;
   Str255            textFormatStr; // "Textual" number format spec
   NumFormatString   formatStr;     // Opaque number format

   // Load the 'itl4' and copy the NumberParts record out of it.
   itl4 = (NItl4Handle)IUGetIntl(4);
   if (itl4 == nil)
      return resNotFound;
   numberParts = *(NumberParts *)((char *)*itl4 +
      (*itl4)->defPartsOffset);

   // Get the format string, convert it to a NumFormatString, and then
   // use it to format the input number.
   GetIndString(textFormatStr, kFormatStrs, theFormat);
   err = Str2Format(textFormatStr, &numberParts, &formatStr);
   if (err != noErr)
      return err;
   err = FormatX2Str(theNum, &formatStr, &numberParts, theString);
   return err;
}
```

Given a currency value, the following routine formats the number and then adds the currency symbol in the appropriate place. This routine assumes that you use a particular number format for currency values, but you can easily modify it to include an argument that specifies the format item in the string list.

```
OSErr FormatCurrency(extended theNum, Str255 theString)
{
   Intl0Hndl   itl0;
   OSErr       err;
   Str255      currencySymbol, formattedValue;

   // First, format the number like this: ##,###.00. FormatX2Str will
   // replace the "," and "." separators appropriately for the font
   // script.
   err = FormatANum(kCurrencyFormat, theNum, formattedValue);
   if (err != noErr)
      return err;

   // Get the currency symbol from the 'itl0' resource. The currency
   // symbol is stored as up to three bytes. If any of the bytes aren't
   // used they're set to zero. So, we use strncpy to copy out the
   // currency symbol as a C string and forcibly terminate it in case it's
   // three bytes long.
   itl0 = (Intl0Hndl)IUGetIntl(0);
   if (itl0 == nil)
      return resNotFound;
   strncpy(currencySymbol, &(*itl0)->currSym1, 3);
   currencySymbol[3] = 0x00;
   c2pstr(currencySymbol);

   // Now put the currency symbol and the formatted value together
   // according to the currency symbol position.
   if ((*itl0)->currFmt & currSymLead) {
      StringCopy(theString, currencySymbol);
      StringAppend(theString, formattedValue);
   } else {
      StringCopy(theString, formattedValue);
      StringAppend(theString, currencySymbol);
   }
   return noErr;
}
```

The 'itl0' resource also includes the decimal and thousands separators. These should be the same values used by FormatX2Str, which gets these symbols from the NumberParts structure in the 'itl4' resource.

If using the extended type in your application makes you queasy, you can easily modify these routines to work with the Fixed type. Just use Fix2X in the FormatX2Str call to convert the Fixed type to extended.

INPUT OF DATES, TIMES, AND NUMBERS

The Script Manager includes routines for parsing formatted values to retrieve a date, time, or number. The process is logically the reverse of formatting a value for output. Most applications don't even deal with formatted numbers. They just read raw numbers (no thousands separators or currency symbols), locate the decimal separator, convert the integer and fraction parts using NumToString, and then put the integer and fraction parts back together.

DEALING WITH CHARACTER ENCODINGS

When writing Macintosh applications, most developers make certain assumptions that cause problems when the application is used in other countries. One of these assumptions is that all characters are represented by a single byte; another is that a given character code always represents the same character. The first assumption causes immediate problems because the two-byte script systems use both one-byte and two-byte character codes. An application that relies on one-byte character codes often breaks up a two-byte character into two one-byte characters, rendering the application useless for two-byte text. The second assumption causes more subtle problems, which prevent the user from mixing text in several different scripts together in one document.

Different versions of the Macintosh system software use a different script by default. Systems sold in the U.S. and Europe use the Roman script. Those sold in Japan, Hong Kong, or Korea use the Japanese, traditional Chinese, or Korean script, respectively. In addition, some sophisticated users have several script systems installed at one time, and System 7.1 makes this even easier. Actually, even unsophisticated users can have two script systems installed at one time. All systems have the Roman script installed, so Japanese users, for example, have both the Japanese and the Roman script available.

For an application to work correctly with any international system software, it must be able to handle different character encodings simultaneously. That is, the user should be able to enter characters in different scripts and edit the text without damaging the associated script information. This section discusses three ways to handle character encodings. These methods require different amounts of effort to implement and provide different capabilities. Of course, those that require the most effort also provide the most flexibility and power for your users. Before we discuss these methods, let's define some more terms.

Language. A human language that's written using a particular script. Several languages can share the same script. For example, the Roman script is used by English, French, German, and so on. It's also possible for the same language to be written in more than one script, although that's a rare exception.

Alphabet, syllabary, ideograph set. A collection of characters used by a language. Some scripts include more than one of these collections. As a simple example, the Roman script includes both an uppercase and a lowercase alphabet. As a more complicated example, the Japanese script includes the Roman alphabet, the Hiragana and Katakana syllabaries, and the Kanji ideograph set. An alphabet, syllabary, or ideograph set isn't necessarily encoded in the same way in two different scripts. For example, the Roman alphabet in the Roman script uses one-byte codes, but the Roman alphabet in the Japanese script uses either one-byte or two-byte codes.

Segment. A subset of an encoding that may be shared by one or more scripts. For example, the simple (7-bit) ASCII characters make up a segment that's shared by all the scripts on the Macintosh. Characters in this segment have the same code in any Macintosh encoding.

Unicode. An international character encoding that encompasses all the written languages of the world. Each character is assigned a unique 16-bit integer. Unicode is a *unified* encoding — all characters that have the same abstract shape share a common character code, even if they're used in more than one language.

METHOD 1: NATIVE ENCODING

The easiest method is to simply pick one character encoding for your localization and stick with it throughout the application. This is usually the native character encoding for the country (and language) that you're targeting with the localized application. For example, if you're localizing an application for the Japanese market, you choose the shift-JIS (Shifted Japanese Industrial Standard) character encoding and modify all your text-handling routines to use this encoding.

The shift-JIS encoding uses both one-byte and two-byte character codes, so you need to use the Script Manager's CharByte routine whenever you're stepping through a string. For a random byte in a shift-JIS encoded string, CharByte tells you if the byte represents a one-byte character, the low byte of a two-byte character, or the high byte of a two-byte character. You also have to handle two-byte characters on input (as described earlier in the section "Keyboard Input") and use the native system and application fonts for text (as described in the section "Avoiding Font Tyranny").

To summarize, the native encoding method has a few advantages:

- It's very easy to implement, so most of your code will work with simple modifications.
- Since you're using the native encoding, the users in the country for which you're localizing will be able to manipulate text using the conventions of the native language.
- Every encoding includes the simple (7-bit) ASCII characters, so they'll also be able to use English.

Unfortunately, this method has many disadvantages:

- You have to create one version of the application for every localization that you do. Each version will use a different native encoding.
- Documents created with one version of the application can't necessarily be used with another version of the application. For example, a document created with the Japanese version that includes two-byte Japanese text will be displayed incorrectly when opened with the French version.
- The user doesn't have access to all the characters in the Roman script (extended ASCII encoding) because these are also used by the native encoding. For example, the ASCII accented characters and extended punctuation use character codes that are also used by the one-byte Katakana syllabary in the shift-JIS encoding. Remember, even the simple international systems really use two scripts — the native script and the Roman script.

METHOD 2: MULTIPLE ENCODINGS

The most complete method for handling character encodings is to keep track of the encoding for every bit of text that your application stores. In this method the encoding (or the script code) is stored with a run of text just like a font family, style, or point size. The first step is to determine which languages you may want to support. Once this is determined, you can decide which encodings are necessary to implement support for those languages. For example, suppose your Marketing department wants to do localized versions for French, German, Italian, Russian, Japanese, and Korean. French, German, and Italian all use the Roman script. Russian uses the Cyrillic script; Japanese uses the Japanese script; and Korean uses the Korean script. To support these languages, you have to handle the Roman, Cyrillic, Japanese, and Korean encodings.

In general, each script that you include requires support for its encoding and, possibly, additional features that are specific to that script. For example, Japanese script can be drawn left-to-right or top-to-bottom, so a complete implementation would handle vertical text. There are other features specific to the Japanese script (amikake, furigana, and so on) that you may also want to implement.

If any of the encodings include two-byte characters, the data structures that you use to represent text runs must be able to handle two-byte codes. When you're processing a text run, the encoding of that run determines how you can treat the characters in the run. For example, you can munge a Roman text run in the usual way, safe and secure in the familiar world of one-byte character codes. In contrast, your dealings with Japanese text runs may be wrought with angst since these runs can include both one-byte and two-byte characters in any combination.

The Script Manager is designed to support applications that tag text runs with a script code. As long as the font of the current grafPort is set correctly, all the Script Manager routines work with the correct encoding for that script. For example, if you specify a Japanese font in the current grafPort, the Script Manager routines assume that any text passed to them is stored in the shift-JIS encoding.

Keyboard script. During this discussion of the multiple encodings method, we've been assuming that you already know the script (and therefore the encoding) of text that the user has entered. How exactly do you know this? The Script Manager keeps track of the script of the text being entered from the keyboard in a global variable. Your application should read this variable programmatically after receiving a keyboard event, as follows:

```
short keyboardScript;

keyboardScript = GetEnvirons(smKeyScript);
```

Once you know the keyboard script, make sure that this information stays with the character as it becomes part of a text run. If the keyboard script is the same as the script of the text run, you can just add this character to the text run. Otherwise, you must create a new text run, tag it with the keyboard script, and place the character in it.

You can also set the keyboard script directly when the user selects text with the mouse or changes the current font. The question is, which script do you set the keyboard to use? That depends on the font of the selected text or the new font the user has

Amikake is a variable shading behind text. Furigana is annotation of a Kanji character that appears in small type above the character (or to the right of the character, in the case of vertical line orientation).•

chosen. The first step is to convert the font into a script and then use the resulting script code to set the keyboard script. This process is known as *keyboard forcing*.

```
short fontScript;

fontScript = Font2Script(myFontID);
KeyScript(fontScript);
```

The user can always change the keyboard script by clicking the keyboard icon (in System 6) or by choosing a keyboard layout from the Keyboard menu (in System 7). As a result, you're no longer sure that the keyboard script and the font script agree when the user actually types something. You should always check the keyboard script against the font script before entering a typed character into a text run. If the keyboard script and the font script don't agree, a new current font is derived from the keyboard script. This process is known as *font forcing*.

```
short fontScript, keyboardScript;

fontScript = Font2Script(myFontID);
keyboardScript = GetEnvirons(smKeyScript);
if (fontScript != keyboardScript)
   myFontID = GetScript(keyboardScript, smScriptAppFond);
```

The combination of keyboard forcing and font forcing is called *font/keyboard synchronization*. Both keyboard and font forcing should be optional; the user should be able to turn these features off with a preferences setting.

Changing fonts. An application that works with multiple encodings must pay special attention to font changes. For each text run in the selection, the application should check the script of the text run against the script of the new font. If the scripts agree, the text run can use the new font. If the scripts don't agree, the application can either ignore the new font for that text run or apply some special processing.

```
short fontScript;
short textRunIndex, textRunCount;

fontScript = Font2Script(myNewFontID);
for (textRunIndex = 0;
      textRunIndex < textRunCount;
      textRunIndex++) {
   if (textRunStore[textRunIndex].script == fontScript)
      textRunStore[textRunIndex].fontID = myNewFontID;
   else
      SpecialProcessing(&textRunIndex, &textRunCount, myNewFontID);
}
```

All the encodings used by the Macintosh script systems include the simple (7-bit) ASCII characters, so it's often possible to convert these characters from one script to another. The special processing consists of these two steps:

1. Breaking a text run into pieces, some of which contain only simple ASCII characters and others that contain all the characters not included in simple ASCII
2. Applying the new font to the runs that contain only simple ASCII characters and leaving the other runs with the old font

```
Boolean FindASCIIRun(unsigned char *textPtr, long textLength,
   long *runLength)
{
   *runLength = 0;

   if (*textPtr < 0x80) {
      // We know that this character is simple ASCII, since values less
      // than 128 can't be the first byte of a two-byte character, and
      // they're shared among all scripts. So, let's block up a run of
      // simple ASCII.
      while (*textPtr++ < 0x80 && textLength-- > 0)
         *runLength++;
      return true;         // Run is simple ASCII.
   } else {
      // We know this character is not simple ASCII. It may be two-byte
      // or it may be some character in a non-Roman script. So, let's
      // block up a run of non-simple-ASCII characters.
      while (textLength > 0) {
         if (CharByte(textPtr, 0) == smFirstByte) {
            // Skip over two-byte character.
            textPtr += 2;
            textLength -= 2;
            *runLength += 2;
         } else if (*textPtr >= 0x80) {
            // Skip over one-byte character.
            textPtr++;
            textLength--;
            *runLength++;
         } else
            break;
      }
      return false;        // Run is NOT simple ASCII.
   }
}
```

```
void SpecialProcessing(short *runIndex, short *runCount,
   short myNewFontID)
{
   TextRunRecord     originalRun, createdRun;
   unsigned char     *textPtr;
   long              textLength, runLength, runFollow;
   Boolean           simpleASCII;

   // Retrieve this run and remove it from the run list.
   GetTextRun(*runIndex, &originalRun);
   RemoveTextRun(*runIndex);

   // Get the pointer and length of the original text.
   textPtr = originalRun.text;
   textLength = originalRun.count;

   // Loop through all of the sub-runs in this run.
   runFollow = *runIndex;
   while (textLength > 0) {
      // Find the length of the sub-run and its type.
      TextFont(originalRun.fontID);
      simpleASCII= FindASCIIRun(textPtr, textLength, &runLength);

      // Create the sub-run and duplicate the characters.
      createdRun = originalRun;  // Same formats.
      createdRun.text = NewPtr(runLength);
      // Real programs check for nil pointer here.
      createdRun.length = runLength;
      BlockMove(textPtr, createdRun.text, runLength);

      // Roman runs can use the new font.
      if (simpleASCII)
         createdRun.fontID = myNewFontID;

      // Add the new sub-run and advance the run index.
      AddTextRun(runFollow++, createdRun);

      // Advance over this sub-run and continue looping.
      textPtr += runLength;
      textLength -= runLength;
   }

   // Dispose of the original run information.
   DisposeTextRun(originalRun);
}
```

Searching and sorting. Applications that work with multiple encodings must also take care during searching or sorting operations. An application that uses only the native encoding can assume that character codes are unique and that any two text runs can be compared directly using the sorting routines in the International Utilities Package. On the other hand, an application that uses multiple encodings must always consider a character code within the context of a text run and its associated script. In this case, character codes are unique only within a script, not across script boundaries, so text runs can't be compared directly using International Utilities routines unless they have the same script. If the script codes are different, the International Utilities routines provide a mechanism for first ordering the scripts themselves (IUScriptOrder).

You have the same problems with searching as with sorting. In addition, search commands usually include options for case sensitivity that could be extended in a multiple encodings application. For example, the Japanese script includes both one-byte and two-byte versions of the Roman characters. For purposes of searching, the user might want to consider these equivalent. The simplified Chinese script also includes both one-byte and two-byte versions of the Roman characters, and these should also be equivalent to Roman characters in the Roman script and in the Japanese script. Just like case sensitivity, considering one- and two-byte versions of a character as equivalent should be an option in your search dialog box.

You can use the Script Manager's Transliterate routine to implement a byte-size insensitive comparison. Use Transliterate to convert both the source and the target text into one-byte characters, then compare the resulting strings. Because all the scripts share the same simple (7-bit) ASCII character codes, this mechanism treats all the Roman characters, both one-byte and two-byte, in every script as equivalent.

Summary. The multiple encodings method has several advantages:

- The user can mix text in any number of scripts within one document.
- You can produce several localized versions of the application from a single code base.
- Users in a particular region can use features intended for users in a different region, even if the product isn't advertised to provide those features.

The disadvantages of this method are apparent from the examples:

- It's much more difficult to implement than the native encoding method.
- The two-byte scripts use mixed one-byte and two-byte encodings, so even though you're keeping track of the script of each text run, you still need to worry about mixed character sizes within a run.

- Because some characters are duplicated between scripts, you need to treat their corresponding character codes as equivalent. This further complicates the basic algorithms you use for text editing, searching, sorting, and so on.

METHOD 3: POOR MAN'S UNIFICATION

Our favorite method combines the power of the multiple encodings method with the simplicity of the native encoding method. The idea is to create a single "native" encoding that encompasses all the scripts included in the multiple encodings method. In the multiple encodings method, some characters are encoded several times: a character can have the same code value in different scripts, or it can have different code values in the same script. For example, the letter *A* has the code value 0x41 in the Roman script and the same one-byte code value in Japanese and traditional Chinese. However, Japanese also encodes the letter *A* as the two-byte value 0x8260, and traditional Chinese also encodes it as the two-byte value 0xA2CF. A unified encoding would map all of the identical characters in the multiple encodings to one unique code value.

You might have noticed that this method has some of the same goals as the Unicode scheme — a single character encoding for all languages with one unique code for every character. Unicode extends this goal to the unification of the two-byte scripts. Characters that have the same abstract shape in the simplified Chinese, traditional Chinese, Japanese, and Korean scripts have been grouped together as a single character under Unicode. Our method doesn't go that far. We unify the simple ASCII characters from all scripts but leave the various two-byte scripts to their unique encodings. Thus the name for this method — poor man's unification.

Segments. The poor man's unification method relies on the concept of a segment. A segment is a subset of an encoding with characters that are all the same byte size. For example, the Roman script is divided into two segments — the simple ASCII segment and the extended ASCII/European segment. The Japanese script has three segments — the simple ASCII segment, the one-byte Katakana segment, and the two-byte segment (including symbols, Hiragana, Katakana, Roman, Cyrillic, and Kanji).

The key to poor man's unification is the simple ASCII segment. This segment is shared among all the scripts on the Macintosh (see Figure 2). Furthermore, poor man's unification treats the various encodings as collections of segments that can be shared among encodings. There's logically only one simple ASCII segment, and all the scripts share it. In the multiple encodings method, characters in this range could be found in each script. That is, the word "Beaker" could be stored in both a Roman text run and in a Japanese text run (as one-byte ASCII). In contrast, an application that uses poor man's unification would tag text runs with the segment, not the script, so these two occurrences of "Beaker" would be indistinguishable.

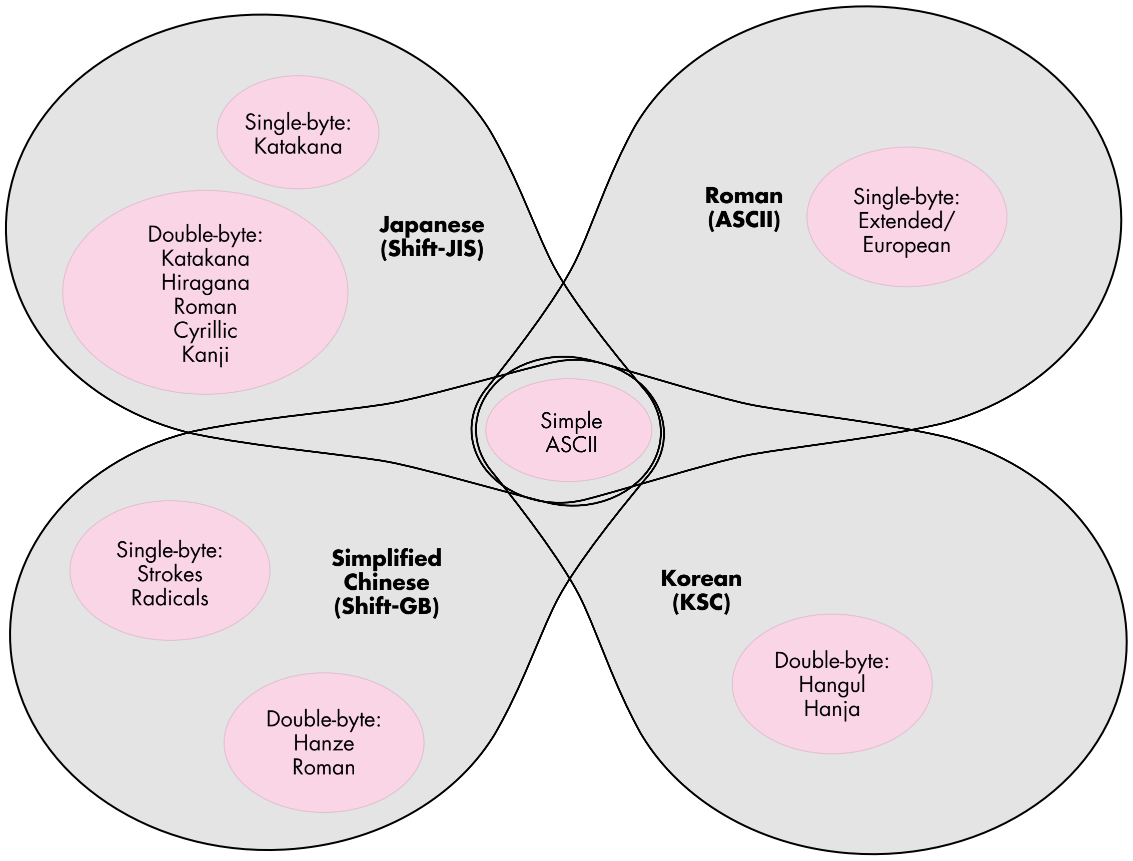


Figure 2
Scripts Sharing the ASCII Segment

The best way to see the advantages of this method is to solve a problem we already considered — changing the font of the selection. With the multiple encodings method, this entailed breaking text runs into smaller runs using our FindASCIIRun routine. With poor man's unification, the same problem is much easier to solve because the runs are already divided into segments and the simple ASCII segment is allowed to take on any font. Other segments are only allowed to use fonts that belong to the same script they do.

```
#define  asciiSegment      0
#define  europeanSegment   1
#define  katakanaSegment   2
#define  japaneseSegment   3
```

```
short fontScript, runSegment;
short textRunIndex, textRunCount;

fontScript = Font2Script(myNewFontID);
for (textRunIndex = 0;
      textRunIndex < textRunCount;
      textRunIndex++) {
   runSegment = textRunStore[textRunIndex].segment;
   if (SegmentAllowedInScript(runSegment, fontScript))
      textRunStore[textRunIndex].fontID = myNewFontID;
}
```

The special processing is gone (surprise). Once you know that a segment isn't included in the script of the font, you can't go any further. Such a segment consists entirely of characters that aren't in the script of the font.

```
Boolean SegmentAllowedInScript(short segment, short script)
{
   switch (script) {
      case smRoman:
         switch (segment) {
            case asciiSegment:
            case europeanSegment:
               return true;
            default:
               return false;
         }
      case smJapanese:
         switch (segment) {
            case asciiSegment:
            case katakanaSegment:
            case japaneseSegment:
               return true;
            default:
               return false;
         }
      default:
         switch (segment) {
            case asciiSegment:
               return true;
            default:
               return false;
         }
   }
}
```

Determining a segment from keyboard input. How do you determine the segment of a character when it's entered from the keyboard?

1. First determine which script the character belongs to by checking the keyboard script.
2. Then use the character-code value and the encoding definitions to assign the character a particular segment.

```
#define ksJISSpace   0x8140

unsigned short keyboardScript;
unsigned short charSegment;
EventRecord    lowByteEvent;

keyboardScript = GetEnvirons(smKeyScript);
charSegment = ScriptAndByteToSegment(keyboardScript, charCode);

if (charSegment == japaneseSegment) {
   // Get low byte of two-byte character from keyboard.
   do {
      // You can get null events between two bytes of a two-byte
      // character.
      GetNextEvent(keyDownMask | keyUpMask | autoKeyMask, &lowByteEvent);
      if (lowByteEvent.what == nullEvent)
         GetNextEvent(keyDownMask | keyUpMask | autoKeyMask,
            &lowByteEvent);
   } while (lowByteEvent.what == keyUp);

   if ((lowByteEvent.what == keyDown) || (lowByteEvent.what == autoKey))
      charCode = (charCode << 8) | (lowByteEvent.message & charCodeMask);
   else
      // We've gotten a valid high byte under the Japanese keyboard
      // with no subsequent low byte forthcoming. Something serious is
      // wrong with the current input method. Return a Japanese space
      // for now. Hmmmm.
      charCode = ksJISSpace;
}

#define kASCIILow    0x00
#define kASCIIHigh   0x7f
#define kRange1Low   0x81
#define kRange1High  0x9f
#define kRange2Low   0xe0
#define kRange2High  0xfc
```

```
short ScriptAndByteToSegment(unsigned short script,
   unsigned char byte)
{
   switch (script) {
      case smRoman:
         if ((byte >= kASCIILow) && (byte <= kASCIIHigh))
            return asciiSegment;
         else
            return europeanSegment;
      case smJapanese:
         if ((byte >= kASCIILow) && (byte <= kASCIIHigh))
            return asciiSegment;
         else if ((byte >= kRange1Low) && (byte <= kRange1High))
            return japaneseSegment;
         else if ((byte >= kRange2Low) && (byte <= kRange2High))
            return japaneseSegment;
         else
            return katakanaSegment;
      default:
         // New scripts and segments added before this.
         return asciiSegment;
   }
}
```

You might think this is quite a bit of effort just to get the low byte of a two-byte character. You're right. And as for Joe's use of the antiquated GetNextEvent instead of the more modern WaitNextEvent, Beaker notes that "It's a cooperative multitasking world and Joe's not cooperating." Joe replies, "Yeah, but I don't want a context switch while I'm trying to get the low byte of a two-byte character."

Changing fonts. Applications that employ poor man's unification still have to worry about font forcing. Here's an algorithm for "smart" font forcing that tries to anticipate which fonts the user will select for text in each segment. When you find a case where the current keyboard script and font script don't agree, instead of using the application font for the keyboard script, search the surrounding text runs for a font that does agree with the keyboard script. Only if you can't find a font that agrees do you default to the application font of the keyboard script. From the user's perspective, this is much nicer. Once the user has selected a font for each script, the application goes back and forth between the fonts automatically as the keyboard script is changed.

```
short fontScript, keyboardScript;

fontScript = Font2Script(myFontID);
keyboardScript = GetEnvirons(smKeyScript);
```

```
// Search backward.
if (fontScript != keyboardScript) {
   for (textRunIndex = currentRunIndex - 1;
         textRunIndex >= 0;
         textRunIndex--) {
      myFontID = textRunStore[textRunIndex].fontID;
      fontScript = Font2Script(myFontID);
      if (fontScript == keyboardScript)
         break;
   }
}

// Search forward.
if (fontScript != keyboardScript) {
   for (textRunIndex = currentRunIndex + 1;
         textRunIndex < textRunCount;
         textRunIndex++) {
      myFontID = textRunStore[textRunIndex].fontID;
      fontScript = Font2Script(myFontID);
      if (fontScript == keyboardScript)
         break;
   }
}

// Punt if we couldn't find an appropriate run.
if (fontScript != keyboardScript)
   myFontID = GetScript(keyboardScript, smScriptAppFond);
```

Applications that use font forcing also have to worry about keyboard forcing. However, if the application includes the feature just described, keyboard forcing is not as important. Many users will prefer to leave the keyboard completely under manual control and allow the "smart" font forcing to choose the correct font when they start typing. The keyboard script is always visible in the menu bar, but the current font is not.

Summary. The poor man's unification method has more advantages than the other two:

- All characters in a run belong to the same segment and therefore take the same number of bytes for their code values. That is, any given run will be all one-byte characters or all two-byte characters, which makes it easier to step through text for deleting or cursor movement. This is in contrast to the multiple encodings method, which can mix one-byte and two-byte characters in a single text run.

THE DEMISE OF ONE-BYTE CHARACTERS

The point of poor man's unification is to simplify your life. On that theme, there's another technique that will help. You can simply decide that *characters are two bytes*. Period. Expand one-byte characters into an unsigned short, with the character code in the low byte and the segment code in the high byte. Then just use unsigned shorts everywhere instead of unsigned chars. You'll find that your code gets easier to write and easier to understand, and that lots of special cases where you would have broken everything out into one- and two-byte cases collapse into one case.

Putting the segment code into the high byte of one-byte characters ensures that the one-byte character codes are unique. If your program handles only one two-byte script, the two-byte codes are also unique. When both these conditions are true, there's no need to store the segment codes in runs, since they're implied by the high byte of each character code.

Here are a few examples using the codes from the sample segments in the section on poor man's unification:

- 0x0041 is the letter *A* in the asciiSegment.
- 0x0191 is the letter *ë* in the europeanSegment.
- 0x8140 is a two-byte Japanese space character.

In other words, one-byte characters carry their segment code in their high byte, and the two-byte characters all belong to the same segment.

You can imagine how much easier searching and sorting algorithms are if you know you can always advance your pointers by two bytes instead of constantly calling CharByte to find out how big each character is. Plus, you might as well get used to storing 16 bits per character, since that's how Unicode works. Yes, it's an extra byte per character — deal with it.

- Runs of simple ASCII and European characters still take one byte per character to store. If you're working on a word processor and plan to keep large amounts of text in memory, this can be an advantage.
- Like the multiple encodings method, this method is easy to extend as you add more scripts to the set your application supports. Each time you add a new script, you need to define the new segments that make up that script and then modify the classification routines to correctly handle the new script and segment codes. Once you locate a specification for the new encoding, these modifications should be straightforward.

Unfortunately, this method has two disadvantages when compared to pure Unicode:

- You still have to deal with one-byte and two-byte characters, even though they won't be mixed together (see "The Demise of One-Byte Characters").
- The application needs to tag each text run with a segment code because the character codes aren't unique across segments.

Unicode does away with both of these disadvantages by making all characters two bytes wide and insisting on one huge set of unique character codes.

PAY ME NOW OR PAY ME LATER

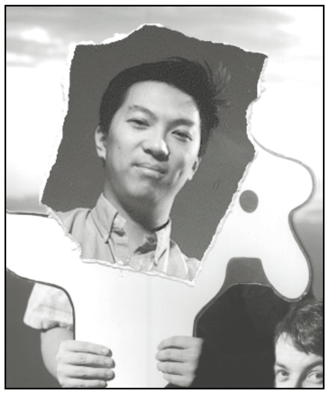
Perhaps the moral of the story is "globalization, not localization," as Joe says. The more generalizations you can build into your application during initial development, the more straightforward your localization process is destined to be, and the less your localized code base will diverge from your original product.

Weigh the size and growth potential of a given language market against the amount of effort required to implement that language. Stick to the markets where your product is most likely to flourish. This article has shown that in some cases you can dramatically reduce code complexity by taking shortcuts — the poor man's unification scheme in this article is a good example. A healthy balance between Script Manager techniques and custom code will help you bring your localized product to market fast and make it a winner.

RECOMMENDED READING

- *Inside Macintosh* Volume VI (Addison-Wesley, 1991), Chapter 14, "Worldwide Software Overview."
- *Inside Macintosh* Volume V (Addison-Wesley, 1986), Chapter 16, "The International Utilities Package," and Chapter 17, "The Script Manager."
- *Inside Macintosh* Volume I (Addison-Wesley, 1985), Chapter 18, "The Binary-Decimal Conversion Package," and Chapter 19, "The International Utilities Package."
- *Inside Macintosh: Text* (Addison-Wesley, 1993).
- *The Unicode Standard, Version 1.0*, Volume 2 (Addison-Wesley, 1992).
- *Macintosh Worldwide Development: Guide to System Software*, APDA #M7047/B.
- *Localization for Japan*, APDA #R0250LL/A.
- *Guide to Macintosh Software Localization*, APDA #M1528LL/B.
- "The TextBox You've Always Wanted" by Bryan K. ("Beaker") Ressler, *develop* Issue 9.

THANKS TO OUR TECHNICAL REVIEWERS
Jeanette Cheng, Peter Edberg, Neville Nason,
Gideon Shalom-Bendor•



JOHN WANG

PRINT HINTS

SYNCING UP WITH COLORSYNC

Apple's recently introduced ColorSync, a color matching software technology, provides a common platform for applications and device drivers to match colors by communicating color information between graphics devices with differing color characteristics. This column starts off with an overview and then delves deeper into the inner workings of ColorSync so that you'll have a better understanding of how to use this new technology. We'll also take a look at how applications and device drivers can take advantage of ColorSync.

WHAT IS COLORSYNC?

ColorSync is an extension to the Macintosh system that's distributed with the Apple Color Printer and the Color OneScanner. It provides a platform for maintaining quality and similarity of images that are moved between different devices. Because different devices typically reproduce different *gamuts* — ranges of colors — ColorSync can be used by applications and device drivers to perform color correction. For example, monitors from different manufacturers have dissimilar gamuts because they use different hardware that drives different cathode ray tubes. In fact, there are minute color differences among the same models due to the video card, internal settings, user adjustments, and even age. ColorSync uses color matching algorithms to visually equate the images produced by different devices. Applications that are ColorSync aware attempt to display a document faithfully on any monitor.

Besides supporting RGB, ColorSync supports color matching with other color spaces, such as CMYK. Printers normally work in the CMYK color space because CMYK colors are subtractive — when added they move the image toward black or dark gray. This is entirely different from RGB monitors, which use additive colors — colors that when added move the image toward white. Consequently, ColorSync is especially useful when it's necessary to match on-screen and printed colors — colors with two very different gamuts.

PROFILES AND COLOR MATCHING METHODS

ColorSync uses two major elements to implement color matching between devices: *profiles* and *color matching methods* (CMMs). The profiles contain the device characterization while the CMMs contain the color matching code to perform the matching. A CMM performs matching between a source profile and a destination profile. A system will have at least one profile for each device to be matched and at least one CMM to perform the matching. Apple ships ColorSync with one Apple CMM and with ColorSync profiles for all Apple monitors currently being manufactured. The open architecture of ColorSync allows third-party developers to create their own profiles and CMMs.

A ColorSync profile is simply a file whose data fork contains a CMProfile record, usually stored in the ColorSync™ Profiles folder. (This folder is in the Preferences folder in your System Folder; your code can get it by calling GetColorSyncFolderSpec.) Profiles may also be stored in a 'prof' resource, as discussed later. A device may have more than one profile; however, only one is selected for use at any given time. For example, printers have profiles for various paper types since the output onto different types of paper can vary. The Apple Color Printer has default profiles for coated paper, transparency film, and plain paper. A monitor may also have several profiles

JOHN WANG (AppleLink WANG.JY) is standing in for Pete ("Luke") Alexander, who was busy working on his QuickDraw GX article for a future issue of *develop.* We expect various members of Developer Technical Support's Printing, Imaging, and Graphics group to take turns writing this column in the future. John also found the time to write his regular QuickTime column; look there (later in this issue) for the real John Wang bio.•

The ColorSync Utilities document on this issue's CD is the comprehensive document that developers should refer to for ColorSync development. However, having worked with many ColorSync developers, I've come across several issues that aren't covered in the ColorSync Utilities document. This column is a conglomeration of hours of discussion and mutual enlightenment.•

for various special gamma settings. ColorSync neither affects nor is affected by the gamma setting. For best results, the user must select a ColorSync profile that matches the gamma.

Here's the data structure for a CMProfile record:

```
typedef struct CMHeader {
   unsigned long     size;
   OSType            CMMType;
   unsigned long     applProfileVersion;
   OSType            dataType;
   OSType            deviceType;
   OSType            deviceManufacturer;
   unsigned long     deviceModel;
   unsigned long     deviceAttributes[2];
   unsigned long     profileNameOffset;
   unsigned long     customDataOffset;
   CMMatchFlag       flags;
   CMMatchOption     options;
   XYZColor          white;
   XYZColor          black;
} CMHeader;

typedef struct CMProfileChromaticities {
   XYZColor    red;
   XYZColor    green;
   XYZColor    blue;
   XYZColor    cyan;
   XYZColor    magenta;
   XYZColor    yellow;
} CMProfileChromaticities;

typedef struct CMProfileResponse {
   unsigned short counts[onePlusLastResponse];
   CMResponseData data[1];
} CMProfileResponse;

typedef struct CMProfile {
   CMHeader                header;
   CMProfileChromaticities profile;
   CMProfileResponse       response;
   IString                 profileName;
   char                    customData[1];
} CMProfile, *CMProfilePtr, **CMProfileHandle;
```

CMMs are components of type 'cmm ' that contain code to perform matching. The component subtype distinguishes between different CMMs. ColorSync ships with the default Apple CMM, which has the subtype 'appl'. Developers who want to provide custom CMMs to perform matching beyond the capabilities of Apple's basic color matching method need to register their CMM subtype with the Apple Registry (AppleLink REGISTRY) to avoid conflict with other CMM manufacturers. The only requirement for subtype naming is that all-lowercase types are not used, because they're reserved by Apple.

A CMM can have six routines, three of which are required:

- CMInit: Given the source and destination profile, prepare to perform color matching.
- CMMatchColors: Match a list of colors using profiles specified by a call to CMInit.
- CMCheckColors: Check a list of colors and determine whether they fall within the gamut of the destination device's color space.

The optional CMM routines are as follows:

- CMMatchPixMap: Match the colors of a pixel map using profiles specified by a call to CMInit.
- CMCheckPixMap: Check a pixel map to determine which pixels fall outside the destination profile's gamut.
- CMConcatenateProfiles: Concatenate two profiles to create one new profile.

WHICH CMM TO USE?

ColorSync profiles that refer to the custom CMMs can be created by setting the CMMType field in the CMHeader to the subtype of the CMM. ColorSync will attempt to use the corresponding CMM when using that profile. However, the custom profiles must still contain the data necessary for compatibility with Apple's default color matching method so that the Apple CMM can be used if the custom CMM is

You make gamma settings in the Monitors control panel by Option-clicking the Options button and, in the dialog box that appears, selecting Use Special Gamma and choosing the special gamma from the pop-up menu.•

Components are described in the Component Manager documentation in the QuickTime Developer's Kit v. 1.5. The information will soon be published in *Inside Macintosh: More Macintosh Toolbox.*•

unavailable. The rules for deciding which CMM to use depend on the source and destination profile:

1. If the source and destination profiles use the same CMM and the corresponding CMM is available, the matching is performed entirely by that CMM. If the CMM is not available, the Apple CMM is used.
2. If the source and destination profiles use different CMMs, then:

a) If the CMM for the destination profile is available, try using that CMM. If the CMM returns an error because it can't perform the color matching, try step b. Since the Apple CMM will never return an error because it's always able to perform matching between two profiles, this is considered a special case, so skip to b.

b) If the CMM for the source profile is available, try using that CMM. If the CMM returns an error because it can't perform the color matching, try step 3. Again, since the Apple CMM will never return an error, this is considered a special case, so skip to step 3.

3. If the CMMs for both the source and destination profiles are available but can't perform the matching as described in step 2, ColorSync matches using the source CMM from the source profile color space to the XYZ color space, and then using the destination CMM from the XYZ color space to the destination profile color space.
4. If step 3 doesn't work because a CMM is missing, the Apple CMM is substituted for the missing one.

COLORSYNC ROUTINES

ColorSync provides high-level and low-level routines that may be used by application and device driver developers. Except for BeginMatching, EndMatching, and DrawMatchedPicture — which are available in System 7 only — the routines are available in system software version 6.0.7 and later. On all systems, ColorSync must be installed. The gestalt selector 'cmtc' returns gestaltColorSync10 (0x0100) for the version of ColorSync that works with system software version 6.0.7, and gestaltColorSync11 (0x0110) for the version that works with System 7. (Note that 6.0.7 must also have version 1.2 of the 32-Bit QuickDraw INIT installed.)

```
// Use Gestalt to get version of ColorSync.
if (Gestalt(gestaltColorMatchingVersion,
        &CMversion) != noErr)
   CMversion = 0;
```

The high-level profile management routines are as follows:

- GetProfile: Get the profile currently selected for a device.
- SetProfile: Add a profile to the device's profile list.
- SetProfileDescription: Set profile description fields for a new profile (typically created by a calibrator).
- GetColorSyncFolderSpec: Get the folder in which ColorSync profiles should be stored.
- GetProfileName: Given a profile, return its name.
- GetProfileAdditionalDataOffset: Given a profile, return the custom data offset.
- ConcatenateProfiles: Concatenate two profiles into one.
- GetIndexedProfile: Return the number of profiles and the profiles from the device's profile list.
- DeleteDeviceProfile: Delete a profile from a device's profile list.

The following high-level matching routines provide a layer of code between application and device driver code and the CMM component code. They simplify color matching by performing matching of all QuickDraw drawing routines.

- BeginMatching: Tell Color QuickDraw to begin matching for the current graphics device using the specified source and destination profiles. (Not available in system software version 6.0.7.)
- EndMatching: Tell Color QuickDraw to stop matching. (Not available in 6.0.7.)

XYZ is a device-independent color space defined by the Commission Internationale de l'Eclairage (CIE). It's an additive color space similar to RGB. Each of the XYZ components is a 1.15-bit unsigned fixed-point number.•

- EnableMatching: Insert picComments to turn matching on or off inside a picture.
- UseProfile: Insert a profile into an open picture.
- DrawMatchedPicture: Draw a picture using color matching. (Not available in 6.0.7.)

These low-level routines perform color matching:

- CWNewColorWorld: Create a color matching world using the specified source and destination profiles.
- CWDisposeColorWorld: Dispose of a color matching world to end the session.
- CWMatchColors: Match a list of colors using the current color matching world.
- CWCheckColors: Check a list of colors to see if they fall within a device's gamut. Use the current color matching world.
- CWMatchPixMap: Match a pixel map using the current color matching world.
- CWCheckPixMap: Check the colors of a pixel map using the current color matching world to determine whether the colors are in the gamut of the destination device.

HOW DOES COLORSYNC WORK?

Now that you have an overview of the basic elements of ColorSync — profiles, CMMs, and routines — we can discuss how ColorSync works by putting all these pieces together.

As mentioned earlier, ColorSync profiles are normally stored in the ColorSync™ Profiles folder in the Preferences folder. In this folder, you'll find a selection of monitor profiles for all Apple color monitor products. In some cases, there are duplicates to account for the color differences between different gamma settings for the monitor. For example, the Apple 16-inch monitor has two profiles: Apple 16" RGB Page-White and Apple 16" RGB Standard. The user selects the profile that corresponds to the Use Special Gamma setting made in the Monitors control panel. This profile — also called the system profile — is selected in the ColorSync control panel. The system profile is used as the default source profile whenever you're matching from a document that doesn't specify a profile or matching to a device that doesn't otherwise have an associated profile.

You may be wondering how to use the ColorSync control panel to select more than one system profile for multiple monitors. Unfortunately, the system profile is an abstraction that shouldn't be associated with any particular device. As described earlier in "Which CMM to Use?" it should be used whenever a profile isn't explicitly specified for a source or destination. ColorSync-aware applications can support multiple monitors by matching to specific graphics devices, thereby overriding the system profile selection. But this isn't recommended except with high-end applications because of difficulties in implementation and complexities for the user.

Applications can determine the current system profile selection with GetProfile. In fact, GetProfile works with any device to get the current profile selection for that device. However, for the call to work, the devices must register their *profile responder.* Every device that uses ColorSync to perform matching must have a profile responder, which is a component that supports the following routines:

- CMGetProfile: Return the profile that the driver would use to perform a match.
- CMSetProfile: Add the profile to the driver's profile list.
- CMSetProfileDescription: Set the device-specific fields in a profile. This allows newly created profiles to be used with the device.
- CMGetIndexedProfile: Get the profile that matches the search criteria.
- CMDeleteDeviceProfile: Delete the profile from the driver's profile list.

The system profile responder is always registered globally in a system, so you can use the ColorSync

Color matching to multiple monitors is implemented by setting the destination profile for each graphics device with SetProfile and then performing matching with DrawMatchedPicture or BeginMatching/EndMatching.•

high-level profile management routines on the system device. Printer driver profile responders are registered only if requested; you register one by calling PrGeneral with the driver opened. The PrGeneral opcode is registerProfileOp (13). By using a profile responder, an application can communicate with any device to request ColorSync profile information. This is especially useful for calibration applications. For example, an application can create a new profile for a printer, call SetProfileDescription to set the device-specific fields in the profile, and then call SetProfile to add the profile to the device driver's profile list.

The following code excerpt demonstrates how to register a device driver profile responder. The complete sample code (including error checking!) is provided on this issue's CD.

```
// Register printer profile responder.
PrOpen();
if ((prError = PrError()) == noErr) {
   printerOpened = true;
   prRecHdl =
      (THPrint)NewHandle(sizeof(TPrint));
   PrintDefault(prRecHdl);

   regProfileBlk.iOpCode = registerProfileOp;
   regProfileBlk.iError = 0;
   regProfileBlk.lReserved = 0;
   regProfileBlk.hPrint = prRecHdl;
   regProfileBlk.fRegisterIt = true;
   PrGeneral((Ptr)&regProfileBlk);
   prError = regProfileBlk.iError;
}
```

You don't see the default profiles for device drivers such as the Apple Color Printer in the ColorSync™ Profiles folder because they're stored as 'prof' resources in the device drivers themselves. However, applications can still create profiles for the printer driver to use by placing them in the ColorSync™ Profiles folder. All printer drivers should search not only in their private profile storage location but in the ColorSync™ Profiles folder as well. In the Apple Color Printer Print Options dialog box, users can choose custom profiles in a pop-up menu if Customized Color Matching is selected. The driver even filters the profiles, so only profiles that match the paper type appear in the menu. This is accomplished by reading in each profile in the folder and searching for the desired values in the CMHeader record. The Apple Color Printer driver stores the profile's paper type in the deviceAttributes field of the profile's CMHeader record. This field is used differently by various devices; for instance, monitor profiles use it to store the gamma setting.

When you finally print to a color printer such as the Apple Color Printer, the printer driver performs matching from the system profile to the printer profile. The application must pass the ColorSync picComments through to the printer for matching to occur. If the application strips out picComments, the printer driver assumes the document uses the system profile. If the picComments contain a custom profile, the printer driver uses that profile as the source profile instead of the system profile. Even a matching method chosen in the Customized Color Matching pop-up menu is overridden by such custom profiles. For example, if a document contains scanned images, the images may have a custom profile that uses photographic matching while the rest of the document uses the solid color system profile.

WHAT DOES AN APPLICATION HAVE TO DO?

In a way, most applications are already ColorSync compatible because they can print to ColorSync-aware printers such as the Apple Color Printer. However, for an application to become ColorSync savvy, it should have three key features:

- It should allow users to tag color matching information to documents and to be able to display them using ColorSync. ColorSync calls such as UseProfile, DrawMatchedPicture, and BeginMatching/EndMatching can be used to do this.
- Applications should allow users to preview the output to a ColorSync-aware printer by matching from the document to the printer profile and back to the system profile. The user can thus view color

Applications that strip picComments from pictures before sending them to the printer driver are not ColorSync compatible because they remove the information that ColorSync uses to perform matching. For general information on picComments, see the Macintosh (Imaging) Technical Note "Picture Comments — The Real Deal" (formerly #91).•

differences that occur in the color matching transition between gamuts. The application can even visually outline colors that can't be displayed faithfully, using the CheckColors routine.

- Most important, the application must preserve picComments in its documents. The application can allow modification of the ColorSync picComments as appropriate, but it must save the information in the document and allow the information to be passed through to the printer.

WHAT DOES A PRINTER DRIVER HAVE TO DO?

A printer driver must first have a responder component that implements the responder routines mentioned earlier. The responder allows ColorSync to communicate with the printer driver. By watching for picComments in the printer port bottleneck procs, the driver is notified of source profile changes and other information as well. The printer driver can then adjust the color matching accordingly.

Matching can be performed with high-level calls such as BeginMatching or with low-level calls such as CWMatchColors. If the printer driver spools pages in the PICT format and uses DrawPicture with an off-screen graphics device for rendering, the high-level calls can be used. Otherwise, matching is best performed with the low-level calls from the QuickDraw bottleneck procs. The Apple Color Printer uses low-level calls and performs color matching in its custom bottleneck procs before rendering occurs. Applications that generate PostScript™ code directly must perform color matching themselves using the low-level calls. They can determine what destination printer profile to use by calling GetProfile.

Apple doesn't ship an updated LaserWriter driver to support ColorSync because it would require a major rewrite of current code. However, applications can work around this by performing the color matching in the application. On the other hand, PostScript Level 2 has color matching support built into the PostScript language, so it would be possible to offload color matching to the PostScript imaging device.

YOUR COLORFUL FUTURE

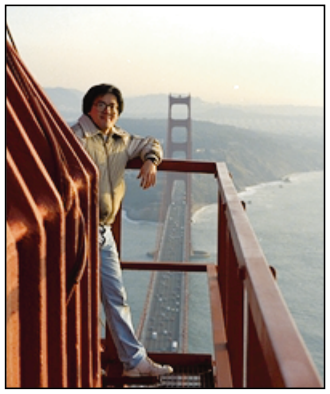
ColorSync is an open architecture platform that enables third-party developers to create profiles, CMMs, and drivers that are mutually compatible. As shown in the past, open architecture promotes market acceptance and user adoption. By using ColorSync as your color matching platform, you're ensured of continued compatibility with future Apple technologies.

As a developer, you can influence the direction of ColorSync; send your feedback to AppleLink DEVSUPPORT. In fact, you can even send me your ColorSync-savvy application (AppleLink WANG.JY) and I'd be thrilled to evaluate it.

Thanks to Bill Guschwan, Tom Mohr, Konstantin Othmer, Steve Swen, and Forrest Tanaka for reviewing this column.•

3-D ROTATION USING A 2-D INPUT DEVICE

One essential part of any 3-D graphics application is the ability to turn an object so that it can be viewed from different sides. This article describes a user interface technique called the Virtual Sphere that allows you to perform continuous 3-D rotation using a 2-D input device such as a mouse. For those who have played with my Rotation Controller application and have been waiting for source code, here it is! For others, this article is a good way to learn something about interactive 3-D graphics and user interfaces for 3-D.


MICHAEL CHEN

There are many situations in which users might want to view a 3-D graphics object from different sides. They might want to do so while constructing an object or rearranging objects in a scene. Or they may be viewing a multimedia document and want to turn around a 3-D object that's embedded in the page. Whatever the context, it's important to provide a simple, intuitive interface for the task that's available to a wide user base.

The problem of 3-D rotation has been approached in many ways. Some people have designed their applications to use higher-degrees-of-freedom input devices such as 3-D mice, 3-D trackballs, the 6-D Spaceball, and the 6-D Polhemus. These devices let you control values for *x*, *y*, and *z* (and perhaps roll, pitch, and yaw) at the same time. Unfortunately, these input devices incur extra cost and must be available on the machine currently being used. The user must also learn how to use the device and possibly learn a new interface paradigm.

Other applications have stayed with 2-D input devices because of their familiarity and availability. However, many of these applications will let you perform rotation only about the *x*, *y*, or *z* axis, while others will let you use the mouse to perform only "2-D rotations," in which the user must specify an axis of rotation lying on, say, the *x*-*y* plane. In both cases, the user needs to change tools or hold down modifier keys (or mouse buttons) to specify rotations about other axes. This is cumbersome, but

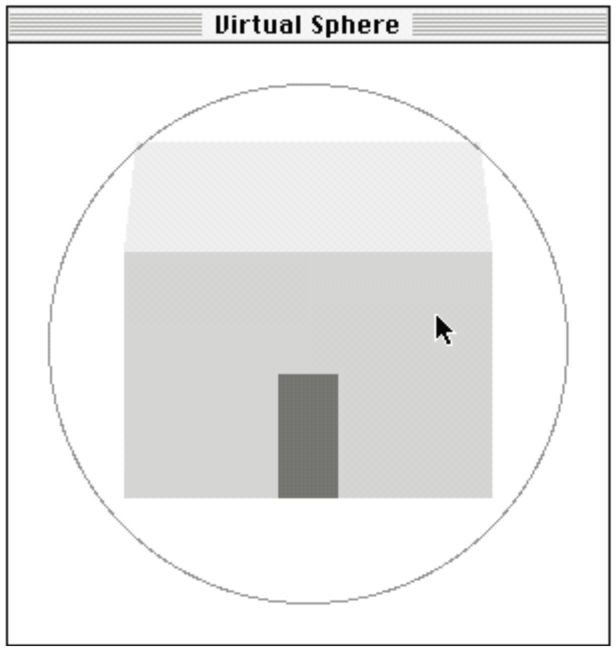
MICHAEL CHEN (AppleLink CHEN.M) works in Apple's Human Interface Group within the Advanced Technology Group. The "high point" of his five years at Apple was digitizing a QuickTime movie on a tower of the Golden Gate Bridge for John Sculley's keynote presentation at Macworld '92 in San Francisco. Michael and his two partners, Dan O'Sullivan and Ian Small, were insured for a total of $12 million for this one-day adventure. Of course, the insurance wasn't for protection against the loss of these valuable Apple researchers; it was to cover the company if one of them were to drop a Macintosh onto a car below. It turned out to be a thrilling but safe trip. When not staring at the computer screen wondering why you should poke everything with this little arrow, Michael enjoys playing flamenco guitar and fooling around with his MIDI toys.

many have accepted the fact that they have only two degrees of freedom when using a device like a mouse.

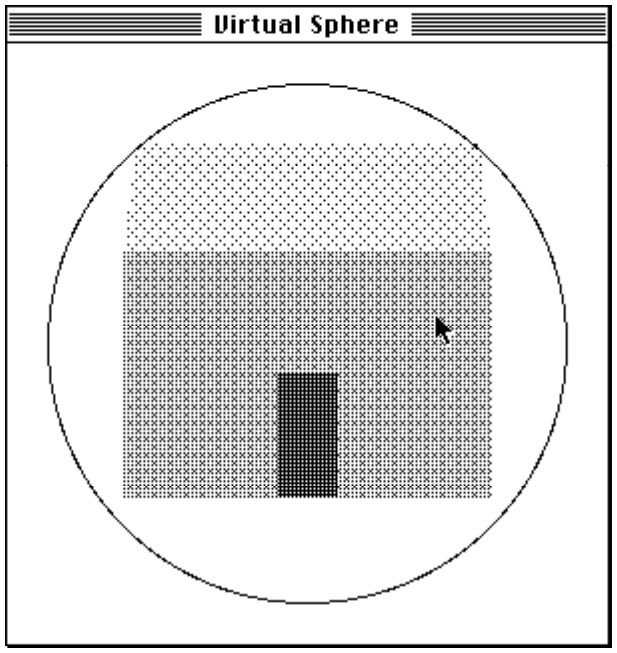
The Virtual Sphere controller is a user interface tool developed to solve the problem of 3-D rotation using a 2-D input device. The controller allows continuous rotation about an arbitrary axis in three-dimensional space. Because the controller works with a 2-D input device, the interface can be used on a wide range of machines with a mouse, trackball, touch screen, or similar device. (This article will assume the use of a mouse.) An important part of the design effort was user testing. Not only must the controller be technologically sound, it must also be easy to learn and use. Testing results will be discussed later.

USING THE VIRTUAL SPHERE INTERFACE

Before getting into the design and implementation of the Virtual Sphere interface, let's first play a bit with the sample application. You'll need to use a machine running System 7 or a System 6 machine with 32-Bit QuickDraw. Find and launch the application VirtualSphereSample, provided on this issue's CD. As shown in Figure 1, you'll see a 3-D house enclosed by the circular Virtual Sphere controller (hereafter called the *cue*).



Grayscale version



Dithered version

Figure 1
The Initial Window, in Grayscale and With Dithering

"Someday, someone is gonna figure out how you can read music without wondering what those tiny little notes are on the page."•

You'll see a color, grayscale, or dithered rendering of the house, depending on your monitor and bit depth. To rotate the house, move the pointer inside the cue and then drag the pointer around. Observe what happens to the house when you drag in the following directions:

- left and right, beginning with the pointer at the center of the cue
- up and down, beginning with the pointer at the center of the cue
- around the edge of or outside the cue

See if you can position the house in a desired orientation.

The Virtual Sphere controller simulates the mechanics of a physical 3-D trackball that can freely rotate about any arbitrary axis in three-dimensional space. The user can imagine the cue to be a glass sphere that's encasing the object to be rotated. Rotation is a matter of rolling the sphere and therefore the object with the pointer. Up-and-down and left-and-right movement at the center of the cue is equivalent to "rolling" the imaginary sphere at its apex and produces rotation about an axis lying on the plane of the screen. Movement along (or completely outside) the edge of the cue is equivalent to rolling the sphere at the edge and produces rotation about the axis perpendicular to the screen.

The Virtual Sphere is unusual in the sense that you seem to be able to squeeze an extra degree of freedom out of a 2-D input device. The action of rolling the Virtual Sphere lets you specify an arbitrary rotation axis in three-dimensional space, with the advantage that you don't need to think about the rotation axis. You just roll, and the object turns in the expected way.

To validate the usefulness of the Virtual Sphere interface, two colleagues and I designed an experiment to compare the performance of different rotational interfaces in a matching task. In the experiment, the computer displays a house at a certain orientation, and the user has to use the given rotational interface to match that orientation. The performance measurement is based on time and accuracy. The result showed that the Virtual Sphere was indeed easy to use and was fastest for complex rotations. If you're interested in the details of the other rotational interfaces and the experiment, they're described in "A Study in Interactive 3-D Rotation Using 2-D Control Devices" (see "Recommended Reading" at the end of this article). A version of the rotation controllers and computer experiment is available on this issue's CD.

HOW THE VIRTUAL SPHERE INTERFACE WORKS

The general idea of this interface is that the Virtual Sphere cue is drawn around the object to be rotated. The cue is centered over the object's center of rotation and should be just large enough to enclose the object. When the user drags over the cue, the successive *x-y* locations of the pointer are used to incrementally rotate the object. The next few paragraphs delve into the mathematics of this process. Those of you with an aversion to vectors and trigonometry may want to skip ahead to the next section.

The orientation of the object is represented in the sample code by a 4 x 4 rotation matrix. At each movement of the pointer, an incremental rotation matrix is computed, using the Virtual Sphere algorithm. This matrix is then concatenated with the object's matrix, and the object is redisplayed. This process is repeated until the user releases the mouse button.

The incremental rotation matrix is computed using the Virtual Sphere algorithm as follows: Figure 2 shows a cue as it appears on the screen and also gives the corresponding 3-D view. The cue is conceptually a hemisphere protruding from the screen. When the pointer is moved from point *p* to point *q* on the screen, we think of it as moving from point *p'* to point *q'* on the surface of the imaginary hemisphere. We compute the points *p'* and *q'* by projecting points *p* and *q* upward (that is, straight out from the screen) from the circle to the surface of the hemisphere. To simplify the math, we'll assume that the cue and hemisphere each have a radius of 1.

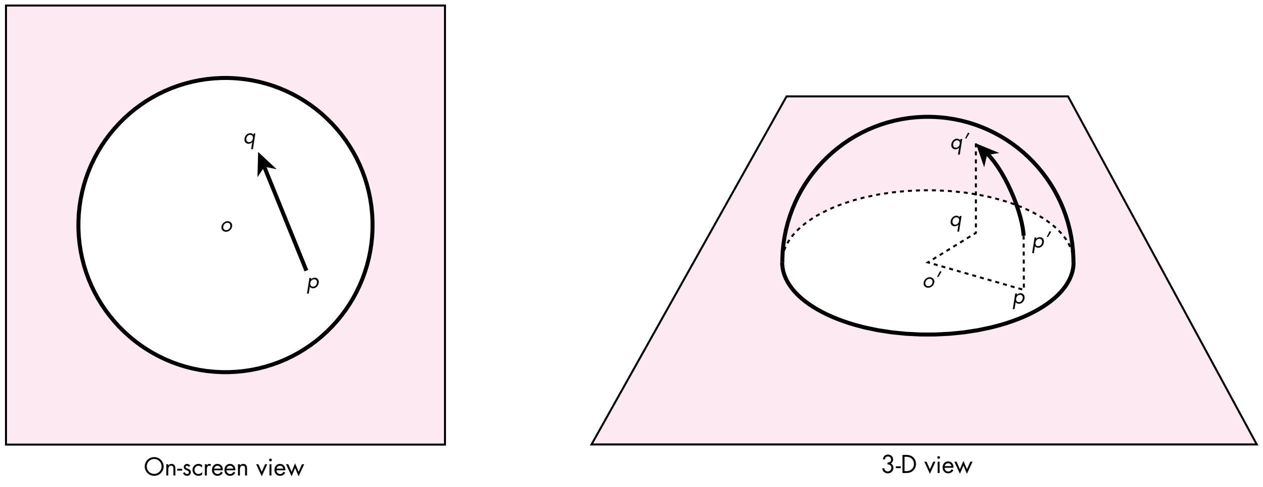


Figure 2
The Virtual Sphere Cue and Its Corresponding 3-D Hemisphere

Given that we now have points *p'* and *q'*, we create vectors from the center of the hemisphere, *o'*, calling them vectors $\overrightarrow{o'p'}$ and $\overrightarrow{o'q'}$. The axis of rotation, $\vec{a}$, is perpendicular to the two vectors and can be computed using the vector cross-product (see Figure 3):

$$\vec{a} = \overrightarrow{o'p'} \times \overrightarrow{o'q'}$$

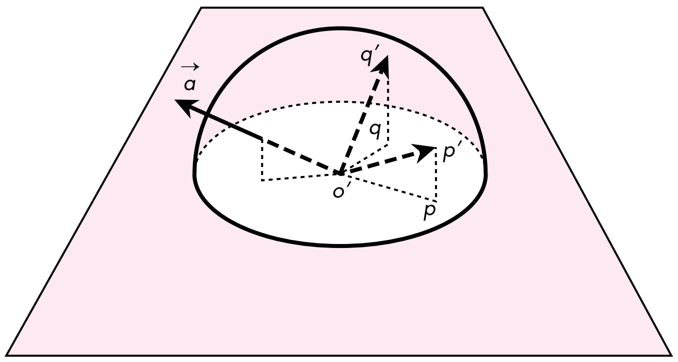


Figure 3
Computing the Axis of Rotation

The amount of rotation, θ, is the angle between the vectors, and is computed from the arc sine of the length of $\vec{a}$:

$\theta = \sin^{-1} |\vec{a}|$

The corresponding 4 x 4 incremental rotation matrix is

$$\begin{bmatrix} ta_x^2+c & ta_xa_y+sa_z & ta_xa_z-sa_y & 0 \\ ta_xa_y-sa_z & ta_y^2+c & ta_ya_z+sa_x & 0 \\ ta_xa_z+sa_y & ta_ya_z-sa_x & ta_z^2+c & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

where

- a_x, a_y, and a_z are the components of $\vec{a}$
- $s = \sin\theta = |\vec{a}|$
- $c = \cos\theta = \overrightarrow{o'p'} \bullet \overrightarrow{o'q'}$ (vector dot-product of $\overrightarrow{o'p'}$ and $\overrightarrow{o'q'}$)
- $t = 1 - c$

Note that we have a choice here for computing the matrix. We can first compute the angle of rotation and then use the sine and cosine functions to obtain s and c. However, knowing vector $\overrightarrow{o'p'}$ and $\overrightarrow{o'q'}$, we can also compute s and c using the length of $\vec{a}$ and the dot-product. The latter approach allows us to compute the matrix efficiently without using any trigonometric calculations, which are expensive.

The Virtual Sphere algorithm just described is an improvement over the one described in the paper "A Study in Interactive 3-D Rotation Using 2-D Control Devices." The paper used a three-step procedure to convert movement of a 2-D input device into a 3-D rotation matrix. However, it turns out that if we first convert the 2-D input to 3-D, the Virtual Sphere calculation can be done much more efficiently.

IMPLEMENTING THE VIRTUAL SPHERE INTERFACE

The preceding description of the Virtual Sphere algorithm probably sounds more complicated than it is. The actual implementation is really quite simple. In fact, the VirtualSphere module contains only one externally visible routine:

```
pascal void VirtualSphere (Point p, Point q, Point cueCenter,
                           Integer cueRadius, Matrix4D rotationMatrix)
{
   CPoint3D    op, oq;

   /* Project mouse points to 3-D points on the +z hemisphere of a unit
    * sphere. */
   PointOnUnitSphere (p, cueCenter, cueRadius, &op);
   PointOnUnitSphere (q, cueCenter, cueRadius, &oq);

   /* Consider the two projected points as vectors from the center of the
    * unit sphere. Compute the rotation matrix that will transform vector
    * op to oq. */
   SetRotationMatrix (rotationMatrix, &op, &oq);
}
```

First, the routine PointOnUnitSphere is used to convert p and q (the previous and current locations of the mouse) into the vectors $\overrightarrow{o'p'}$ and $\overrightarrow{o'q'}$. The routine SetRotationMatrix then computes the 4 x 4 incremental rotation matrix as described in the previous section. The parameters cueCenter and cueRadius define the location of the cue circle in the window. Integer is a macro for long, and CPoint3D is a structure of three doubles.

USING THE VIRTUAL SPHERE INTERFACE

Let's look at how we use the VirtualSphere routine to rotate a 3-D object interactively.

```
void DoRotation (WindowPtr window, EventRecord *event, Matrix4D
                 objectMatrix)
{
   Point       p, q;
   short       dx, dy;
   Point       sphereCenter;
   Integer     sphereRadius;
   Matrix4D    tempMatrix;
   Matrix4D    rotationMatrix;

   p = event->where;
   GlobalToLocal (&p);     /* Get mouse-down point in local coordinates.*/

   /* Figure out where to place the Virtual Sphere cue. */
   sphereCenter.h = kSphereCenterH;
   sphereCenter.v = kSphereCenterV;
   sphereRadius = kSphereRadius;
   while (StillDown()) {
      GetMouse (&q);
      dx = q.h - p.h;
      dy = q.v - p.v;
      if (dx != 0 || dy != 0) {
         VirtualSphere (p, q, sphereCenter, sphereRadius, rotationMatrix);
         MultiplyMatrix (objectMatrix, rotationMatrix, tempMatrix);
         CopyMatrix (tempMatrix, objectMatrix);
         DrawWindow (window);       /* Update the window. */
         p = q;   /* Remember previous mouse point for next iteration.*/
      }
   }
}
```

When DoRotation is called, the 3-D object's current matrix is passed in as objectMatrix. In the sample application, the Virtual Sphere cue is always centered on the window and has a fixed size. Thus, sphereCenter and sphereRadius are assigned with constants. In a general application, you'll need to determine which object is selected and figure out the size and location of the cue (in the window's coordinates) to surround the object. In any case, while the mouse button is still down and the mouse has moved, we call VirtualSphere to obtain the incremental rotation matrix. This matrix is concatenated onto the 3-D object's current matrix using MultiplyMatrix and CopyMatrix. We then redraw the window to display the object at its new orientation. This process is repeated until the mouse button is released.

CREATING A SIMPLE 3-D GRAPHICS SYSTEM

The real point of this article and the sample code is to demonstrate the Virtual Sphere interface. However, it turned out that a large part of the effort involved went into creating a simple 3-D graphics system. It was much more work than the implementation of the Virtual Sphere algorithm itself! I wanted to provide a simple system that would be accessible by the majority of Macintosh programmers and would allow even low-end Macintosh models to do interactive 3-D graphics.

The graphics system I came up with is based on Graf3D. Graf3D is a simple library for drawing 3-D graphics using a fixed-point interface to QuickDraw's integer coordinates. I chose Graf3D because it's included in the THINK C and MPW environments. It uses fixed-point math and runs reasonably quickly even on a Macintosh SE! This means that the sample code should be quite usable for all *develop* readers.

One major caveat is that the Graf3D library is unsupported. Most people doing 3-D graphics on the Macintosh probably won't care about this point, because they write their own 3-D software and will simply port the Virtual Sphere code to their system. I've provided the sample code using Graf3D to show how the whole system works. Also, it's nice to be able to show that Graf3D is not as brain dead as some people might think.

GRAPHICS SYSTEM SPECIFICATION

The graphics system we need is extremely simple. It needs to be able to display only one relatively simple 3-D object at the center of a window. We'll assume that the entire object is visible so that we won't have to worry about 3-D polygon clipping (which, unfortunately, is left as an exercise for the students in most 3-D graphics courses). The object should be displayed with perspective projection. The displayed object can only be rotated. We predefine the center of rotation to be the center of the object.

On the display screen, we define the origin of the 3-D coordinate system at the center of the screen, the *x* axis as extending to the right, the *y* axis as extending upward, and the *z* axis as coming out of the screen toward the viewer. Note that in the QuickDraw coordinate system, the *y* axis extends in the opposite direction.

For the sake of cosmetics, the graphics system should adapt to the monitor bit depth so that the graphics can be shown on color, grayscale, and black-and-white displays. The system must use double buffering to eliminate screen flicker.

To make the Virtual Sphere implementation easier to understand, I've used floating-point math to compute the axis of rotation, the angle of rotation, and the 4 x 4 rotation matrix. However, we'll take advantage of the fixed-point math used in Graf3D to speed up graphics drawing.

SETTING UP GRAF3D

To implement our simple graphics system using Graf3D, we create and associate a Port3D to the grafPort in which the 3-D object is to be displayed. We place the camera at some distance on the positive *z* axis, looking at the origin. We draw the object centered at the origin. With this setup, we have the 3-D view as specified above.

EXTENDING GRAF3D TO DRAW POLYGONS

Graf3D provides only two calls, MoveTo3D and LineTo3D, to perform line drawings in 3-D. These calls are analogous to the 2-D MoveTo and LineTo calls, except that the 3-D calls require an additional *z* parameter. Graf3D doesn't have calls to draw 3-D polygons. However, that doesn't mean we have to write our own routine from scratch. We'll take advantage of the fact that MoveTo3D and LineTo3D, after performing the math to project 3-D onto 2-D, will call MoveTo and LineTo to draw the line on-screen (of course, it helps to have access to the source code for Graf3D). Hence, drawing a polygon projected from 3-D is no more difficult than drawing a regular 2-D polygon: we use the standard QuickDraw polygon routines. Here's an example that draws a filled 3-D triangle:

```
polyHdl= OpenPoly ();
   MoveTo3D (Long2Fix(0),Long2Fix(0),Long2Fix(0)); /* 1st point */
   LineTo3D (Long2Fix(2),Long2Fix(5),Long2Fix(0)); /* 2nd point */
   LineTo3D (Long2Fix(5),Long2Fix(1),Long2Fix(0)); /* 3rd point */
   LineTo3D (Long2Fix(0),Long2Fix(0),Long2Fix(0)); /* 1st point again */
ClosePoly ();
PolyColor (&rGBColor);
FillPoly (polyHdl, lgPolyShade);
KillPoly (polyHdl);
```

PolyColor is a new routine for specifying the color of the polygon; lgPolyShade specifies the polygon's fill pattern. These two items are explained in the next section.

DEALING WITH BLACK-AND-WHITE AND COLOR QUICKDRAW

In general, if you want to take advantage of a grayscale or color display when available, you have to write parallel code. You also need to worry about different versions of QuickDraw so that you don't make the mistake of making Color QuickDraw calls on machines with black-and-white QuickDraw. (See the Graphical Truffles column in this issue for a discussion of the different possible QuickDraw versions.) In our simple graphics system, we want to be able to draw in color when we can and draw in simulated grays using dither patterns when we have a 1-bit display. Parallel code is eliminated by hiding all the complexity of different QuickDraw versions inside the routine PolyColor.

```
static ConstPatternParam   lgPolyShade;
pascal void PolyColor (const RGBColor *rGBColor)
{
   if (gDrawInColor) {
      lgPolyShade = qd.black;
      RGBForeColor (rGBColor);
   } else {
      /* Convert rGBColor to a dither pattern. */
      unsigned long index;
      index = RGBToGrayscale (rGBColor, (**lgDitherPatterns).patListSize);
      lgPolyShade = (**lgDitherPatterns).patList [index];
      ForeColor (blackColor);
   }
}
```

PolyColor takes an RGBColor as an argument. If we're drawing in color (the global variable gDrawInColor is true), PolyColor simply calls RGBForeColor, and lgPolyShade is set to a black pattern. When FillPoly(polyHdl, lgPolyShade) is eventually called, the polygon will get drawn in that color. If we're drawing in black and white, PolyColor makes the foreground color black and converts the RGB value to one of 65 dither patterns (including white), which is assigned to lgPolyShade (see "Converting RGB Color to a Grayscale Value"). When FillPoly(polyHdl, lgPolyShade) is called, the polygon will be filled with that dither pattern. Note that lgDitherPatterns is just a (locked) handle to a PAT# resource. We're limited to 65 possible dither patterns because the Pattern data structure is 8 x 8.

CONVERTING RGB COLOR TO A GRAYSCALE VALUE

The conversion from an RGB value to a grayscale value can be done in a number of ways. The most obvious way is to have the R, G, and B components each contribute equally to the gray value. However, this implies that, for example, pure red, green, and blue colors will all map to one value, which might not be desirable. In the sample code, the conversion routine RGBToGrayscale employs a set of often-used weighting factors for the RGB components. The routine returns an integer value from 0 to maxGrayValue–1:

```
unsigned long RGBToGrayscale (const RGBColor *rGBColor, Integer
                              maxGrayValue)
{
   #define  kRWeight       3
   #define  kGWeight       6
   #define  kBWeight       1
   #define  kTotalWeight   (kRWeight + kGWeight + kBWeight)

   unsigned long intensity;
   unsigned long index;
   intensity = kRWeight*rGBColor->red + kGWeight*rGBColor->green +
               kBWeight*rGBColor->blue;
   index = intensity * maxGrayValue/ kTotalWeight / (USHRT_MAX+1);
                        /* Note integer math. Order matters. */
   return (index);
}
```

DEALING WITH ROTATION

In Graf3D, the Port3D data structure contains a 4 x 4 xForm matrix that defines how the object is to be transformed before it's displayed. Recall that in our DoRotation routine we compute the object's rotation matrix directly. Thus, all we need to do is copy this matrix to xForm before we display the object. However, we need to do some number conversions since the rotation matrix is in floating point and the xForm matrix is in fixed point. The routine Matrix2XfMatrix does the necessary conversion:

```
pascal void Matrix2XfMatrix (Matrix4D fromMatrix, XfMatrix toMatrix)
{
   Integer i, j;
   for (i=3; i>=0; i--) {
      for (j=3; j>=0; j--) {
         toMatrix[i][j]= X2Fix (fromMatrix[i][j]);
      }
   }
}
```

DOUBLE BUFFERING

The sample code contains a module for off-screen drawing that uses GWorlds to eliminate drawing flicker. The use of GWorlds means that graphics acceleration comes for free if it's available in hardware. This module provides a very simple way of dealing with GWorlds. It contains only five routines:

```
pascal OSErr   InitializeOffscreen (Boolean *gWorldAvailable);
pascal void    FreeOffscreen (GWorldPtr offscreenGWorld);
pascal QDErr   CheckOffscreenForWindow (GWorldPtr *offscreenGWorld,
                                     short pixelDepth, WindowPtr window);
pascal void    BeginDrawingOffscreen (GWorldPtr offscreenGWorld,
                                     WindowPtr window);
pascal void    EndDrawingOffscreen (GWorldPtr offscreenGWorld,
                                     WindowPtr window);
```

The routine InitializeOffscreen determines whether GWorlds are available on a particular machine and internally remembers whether System 6 or 7 is running. The latter is needed because there are subtle differences between the GWorld calls in System 6 and those in System 7 (see Offscreen.c and *Inside Macintosh* Volume VI, page 21-19).

The routine CheckOffscreenForWindow checks to see whether a GWorld has the proper bit depth and memory boundary aligned for efficient transfer using CopyBits. If a GWorld hasn't been allocated, a new one is created. If an existing GWorld has the wrong bit depth or isn't memory aligned, it's reallocated. This routine hides the subtle differences between the Toolbox NewGWorld and UpdateGWorld calls. You should call this routine when a window has just been created *and* whenever you think the GWorld is out of sync with the screen.

The routine BeginDrawingOffscreen redirects drawing to the GWorld. The routine EndDrawingOffscreen ends the redirection and copies the off-screen buffer onto the window. The routine FreeOffscreen frees the GWorld when it's no longer needed.

The basic calling sequence for this module is as follows:

```
InitOffscreen (...);
window = GetNewWindow (...);
gWorld = nil;
CheckOffscreenForWindow (&gWorld, window, ...);
. . .
while (still not done with drawing) {
   CheckOffscreenForWindow (&gWorld, window, ...);
   BeginDrawingOffscreen (&gWorld, window);
   /* Draw something */
   EndDrawingOffscreen (&gWorld, window);
}
FreeOffscreen (&gWorld);
```

In the actual code, CheckOffscreenForWindow is called only when there's an update event that could have been generated when the user changed the monitor bit depth. It's not necessary to call CheckOffscreenForWindow during the loop when the mouse is interacting with the object.

CODE OPTIMIZATION

I didn't optimize the sample code because it would have detracted from presenting a clear implementation of the Virtual Sphere algorithm and the 3-D graphics system. For example, we deal only with 3-D rotation in this program, so we don't really need to have a general 4 x 4 matrix when a 3 x 3 matrix would do. Even if we were willing to waste storage, some of the math routines (such as CopyMatrix, MultiplyMatrix, SetRotationMatrix, and Matrix2XfMatrix) could have been optimized to use only the upper left 3 x 3 cells of the 4 x 4 matrix. It might also be worthwhile to do a full fixed-point implementation of the Virtual Sphere algorithm.

MPW C VERSUS THINK C

One of my objectives in creating this sample code was to make sure that it could be used in both the MPW and THINK C environments. I also wanted to allow the option of compiling the application using SANE or a hardware floating-point unit (FPU). This turned out to be a learning experience in itself. Here are a few things I picked up from the process:

- The main difference between the two environments is the way in which floating-point numbers are handled. MPW provides an easy way of switching between SANE and a hardware FPU simply by switching compile flags and by including the proper versions of the math libraries. The header files need not be changed. With THINK C, the SANE and ANSI math libraries aren't integrated. You must include either SANE.h or Math.h, but not both.
- In THINK C, some of the transcendental functions (for example, asin and atan2) are not available when SANE is used.
- In THINK C, there are five floating-point formats. Some of the fixed-point/floating-point conversion routines (such as X2Fix and Fix2X) are incompatible when native floating-point format is used.

I created the files MyMath.h and MyMath.c to hide all the ugliness of floating-point math from the rest of the code.

Compiling the code in both environments requires more careful coding. A version that compiled fine in one environment would get errors and warnings in the other. In general, MPW C is a bit pickier about type checking.

On my wish list is a common pragma structure for changing compile options in the source code. I wanted to display an error message if the user launched a version of the sample application compiled for a hardware FPU on a non–floating-point machine. Here's an example of the gymnastics I had to go through to make sure the routine MessageAlert could be executed on all processors:

```
#ifdef applec
#pragma push                 /* MPW: save compiler flags              */
#pragma processor 68000      /* Generate 68000 instructions only      */
#endif
pascal void MessageAlert (Str255 message)
{
   #ifdef THINK_C            /* THINK C: Generate 68020 instructions...*/
   #pragma options(!mc68020)  /* NOT! Silly way of saying 68000        */
   #endif                    /* instructions only. Note this pragma   */
                             /* is defined only until end of routine.  */
   SetCursor (&qd.arrow);
   ParamText (message, "", "", "");
   SysBeep (10);
   (void) Alert (rMessageAlert, nil);
}
#ifdef applec
#pragma pop                  /* MPW: restore compiler flags           */
#endif
```

Why should the processor pragma matter for such a simple routine? The reason is that MPW C with the -68020 flag will generate an RTD (not available on a 68000 machine) instead of an RTS instruction for returning to the calling routine. This would cause the resulting sample application to crash on a 68000 machine instead of putting up an alert message. THINK C doesn't seem to generate the RTD instruction even when asked to generate 68020 code. However, I put the pragma in for THINK C just in case.

GOING ON FROM HERE

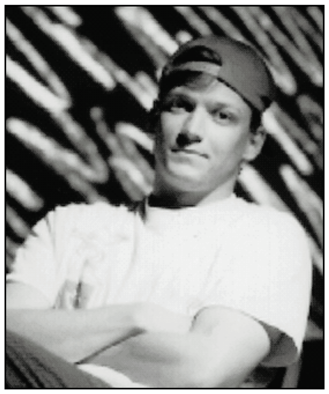
I hope this article has whetted your appetite for 3-D graphics and 3-D user interfaces. Several companies have released commercial programs that improve on the Virtual Sphere concept. Silicon Graphics' Inventor Toolkit contains a version with three orthogonal "ribbons" around the sphere to provide constrained-axis rotation. Virtus Walkthrough contains a version with momentum — the object continues to spin in the direction of the pointer movement when the mouse button is released.

The Virtual Sphere interface introduces a new interaction technique that was backed with user testing. If you're doing new interface development, I encourage you to use the same process: design followed by testing with iterations. Remember to keep the big picture in mind. 3-D rotation is just one small task that the user has to do in a 3-D application. All the interaction techniques for manipulating objects must work together and must be appropriate for the 3-D task the user wants to perform.

RECOMMENDED READING

- "A Study in Interactive 3-D Rotation Using 2-D Control Devices" by Michael Chen, S. Joy Mountford, and Abigail Sellen (*ACM Siggraph '88 Proceedings,* Volume 22, Number 4, August 1988, pages 121–129). An electronic version of this paper is available on this issue's CD.
- "A Technique for Specifying Rotations in Three Dimensions Using a 2-D Input Device" by Michael Chen and K. C. Smith (*Proceedings IEEE Montech '87 — Compint '87*, November 1987, pages 118–120).
- "An Object-Oriented 3D Graphics Toolkit" by Paul Strauss and Rikk Carey (*ACM Siggraph '92 Proceedings,* Volume 26, Number 2, July 1992, pages 341–349).
- "Tablet Based Valuators That Provide One, Two or Three Degrees of Freedom" by K. B. Evans, Peter P. Tanner, and M. Wein (*ACM Siggraph '81 Proceedings,* Volume 15, Number 3, August 1988, pages 91–97).
- "Iterative Design of an Interface for Easy 3-D Direct Manipulation" by Stephanie Houde (*ACM CHI '92 Proceedings*, May 1992, pages 135–142).
- "Three-Dimensional Widgets" by B. Conner, S. Snibbe, K. Herndon, D. Robbins, R. Zeleznik, and A. Van Dam (*Proceedings of the 1992 Symposium on Interactive 3D Graphics,* pages 183–188).
- "Understanding Graf3D" by Scott Berfield (*The Essential MacTutor* Volume 3, pages 230–238).
- *Computer Graphics*: *Principles and Practice*, 2nd ed., by J. D. Foley, A. Van Dam, S. K. Feiner, and J. F. Hughes (Addison-Wesley, 1990).

THANKS TO OUR TECHNICAL REVIEWERS
Michael Chmilar, Forrest Tanaka, Dan Venolia•



BILL GUSCHWAN

GRAPHICAL TRUFFLES

FOUR COMMON GRAPHICS ANSWERS

QuickDraw regions were almost lost when Bill Atkinson crashed his car and nearly killed himself. Considering that Steve Wozniak also crashed his airplane, crashing must be a hallmark of Apple genius. I'm no Macintosh wizard, though I did crash my car once. To aid and abet other non-wizards, I'll divulge four cool answers that apply to nearly any QuickDraw question:

- Check the graphics state.
- Check the QuickDraw version.
- Do it off-screen.
- Use the bottlenecks.

They may not answer your questions completely, but they'll probably get you partway there.

CHECK THE GRAPHICS STATE

Whenever you call a QuickDraw routine, its behavior depends heavily on the state of the machine at the time of the call: things like the pen size, transfer mode, and color environment all affect the drawing. Most of the state information QuickDraw depends on can be found in two handy locations — the current grafPort and the current GDevice.

The grafPort maintains state information for the pen, the text, and the bitmap (or pixel map) to draw into. The GDevice defines the color environment, among other things. This information is accessed by many QuickDraw routines. For example, the LineTo routine draws a line from the current pen location to the point you pass to the routine, using the current pen size, pattern, and transfer mode; all these values are fields of the current grafPort. Because most QuickDraw routines use these "implied" parameters, you can't fully understand the behavior of a QuickDraw routine without knowing about them.

The current grafPort also defines where your drawing will occur. Even though you call a routine, it may not draw, because QuickDraw applies a rectangle — the portRect — and two regions — the visRgn and clipRgn — to your drawing. No drawing will occur outside the intersection of these areas. QuickDraw places control of the clipRgn in your hands, first initializing it to be wide open (a rectangular region that covers the entire QuickDraw coordinate space). If your grafPort is in a window, control of the visRgn is placed in the hands of the Window Manager. (A region is a truly marvelous concept, a compact description of strange shapes that can be extended and changed dynamically. It's a good thing Bill Atkinson survived his crash.)

Let's try applying this first answer to a common QuickDraw question: Why can't you nest calls to OpenPicture? Well, as you may know, when you call OpenPicture to begin saving picture data, a handle is created to store the picture information until the corresponding ClosePicture call. This handle is kept in the picSave field of the current grafPort. If you nest a second OpenPicture call, where in the grafPort will you store the newly created handle? Answer: There is no place, so you can't nest OpenPicture calls.

Because so many other Managers rely on QuickDraw, this answer will help with questions about other Managers as a bonus.

CHECK THE QUICKDRAW VERSION

There are currently seven versions of QuickDraw. You can find out which version is available using Gestalt, and that's usually the most important thing for your code to know about. But many developers also

BILL GUSCHWAN (AppleLink ANGUS) asked Howard Roark to dialog with him: "So, Angus, you ditched med school to become a protector of the dogcow? I love you, man." Angus: "Well, Howard, as Tori Amos would say, 'Sometimes I hear my voice and it's been here silent all these years.'" Howard: "You know, I ditched architecture school, not unlike David Byrne. Speaking of Talking Heads, you got kicked out of one of his concerts because you wanted to dance." Angus: "Words are very unnecessary, they can only do harm, so I dance." Howard: "Even your idol Wittgenstein went back to school. What about you?" Angus: "As you know, Howard, even Atlas shrugged."•

want to know which machine and system software combinations produce which versions of QuickDraw. For example, some developers code for 32-Bit QuickDraw and want to inform their users of the minimum Macintosh machine requirement. The ROM version, extensions, and system software all combine to affect which version of QuickDraw is available.

ROM versions of QuickDraw can be neatly categorized into three classes of machines: black and white, Color QuickDraw, and "ci class." The original Macintosh and the Macintosh 512K, Plus, Portable, SE, and PowerBook 100 are examples of the black-and-white class. The Macintosh II, IIx, IIcx, and SE/30 fall into the Color QuickDraw (256K ROM) class. The Macintosh IIci, IIsi, LC II, IIfx, and later models belong to the ci class (>256K ROM).

Black-and-white class. There are only two common versions of QuickDraw on black-and-white machines today: original black-and-white QuickDraw and System 7 black-and-white QuickDraw. (The uncommon ones are present only with pre-Macintosh Plus ROMs or system versions earlier than 4.2.) When System 7 is installed on a machine of this class, it installs some new routines so that a few Color QuickDraw routines can be used (you get 1-bit GWorlds, you can correctly display pictures containing direct-color information, you can create version 2 pictures, and so on).

Black-and-white QuickDraw is documented in *Inside Macintosh* Volumes I and IV. For a comprehensive list of the routines that System 7 adds to black-and-white QuickDraw, see "QuickDraw's CopyBits Procedure: Better Than Ever in System 7.0" in *develop* Issue 6.

Color QuickDraw class. This class of machines has 8-bit Color QuickDraw built into ROM, so it will always be there regardless of the system version. When these machines are running system versions earlier than System 7, they can be extended to handle direct color through the use of the 32-Bit QuickDraw INIT. Finally, if they're running System 7, System 7 Color QuickDraw is available.

Inside Macintosh Volume V describes 8-bit Color QuickDraw. For documentation on the various 32-Bit QuickDraw versions, including System 7 Color QuickDraw, the best place to look is *Inside Macintosh* Volume VI. If you really need to know the differences in capabilities among the versions, 32-Bit QuickDraw v. 1.0 is covered in "Realistic Color for Real-World Applications" in *develop* Issue 1, and the features added in 32-Bit QuickDraw v. 1.2 are documented in the Tech Note "32-Bit QuickDraw: Version 1.2 Features."

ci class. This class of machines has only three possible QuickDraw versions. The least common version, 32-Bit QuickDraw v. 1.01, is found on a IIci running system software version 6.0.4. The other machines in this class that can run System 6 need at least version 6.0.5, which will patch in 32-Bit QuickDraw v. 1.2. Finally, System 7 provides its own version of Color QuickDraw.

Again, for documentation on the various 32-Bit QuickDraw versions, including System 7 Color QuickDraw, see *Inside Macintosh* Volume VI.

In the GestaltEqu.h header file, you'll find Gestalt values for six QuickDraw versions:

```
gestaltOriginalQD =  0x000,   // 1-bit QD
gestalt8BitQD =      0x100,   // 8-bit color QD
gestalt32BitQD =     0x200,   // 32-bit v1.0
gestalt32BitQD11 =   0x210,   // 32-bit v1.1
gestalt32BitQD12 =   0x220,   // 32-bit v1.2
gestalt32BitQD13 =   0x230,   // 32-bit v1.3
```

One of these — gestalt32BitQD11 — will never be returned, so this list accounts for only five of the total of seven versions. The sixth is 32-Bit QuickDraw v. 1.01, mentioned above, which returns the Gestalt value 0x201 but doesn't have a gestalt constant defined for it. The seventh is System 7 with a black-and-white machine: You'll need to check for both black-and-white QuickDraw (gestaltOriginalQD) and System 7 (gestaltSystemVersion greater than or equal to $0700). If both are true, you're running System 7 black-and-white QuickDraw. That's the only way to tell.

Table 1 shows all the possible combinations, in one handy location.

Exactly which permutations you need to code for depends entirely on what you're doing, but typically the major divisions are color versus black-and-white, direct color versus indexed color, and GWorlds versus no GWorlds. Whenever possible, of course, you should make decisions in your code based on the QuickDraw version rather than on the specific machine configuration.

DO IT OFF-SCREEN

Off-screen environments give you explicit and total control over an image. Since the image and its associated data structures are no longer tied to a physical device, many of the complexities and limitations of QuickDraw are reduced, and your hands — previously tied tightly behind your back — are now freed. You'll typically manipulate your image off-screen and then use CopyBits to move the image to the screen. The FX snippet on this issue's CD provides a robust demonstration of some snazzy CopyBits effects, and there's a nice overview of how to perform animation using off-screen graphics environments in the Graphical Truffles column ("Animation at a Glance") in *develop* Issue 12.

Using CopyBits and off-screen environments for speed is covered eloquently in Konstantin Othmer and Mike Reed's article "Drawing in GWorlds for Speed and Versatility" in *develop* Issue 10, so I won't dwell on it here. Also, the Tech Note "Principia Offscreen Graphics Environments" gives details for creating off-screen environments on machines without 32-Bit QuickDraw (see the discussion above).

The point is this: when faced with a question in the "How do I . . ." category, try this answer on for size first. That may be as far as you need to go.

Table 1
Possible Combinations of ROM Versions and System Software Versions

ROM Class	System Version	Gestalt Value
Black-and-white class	Pre-7.0	gestaltOriginalQD
	7.0 and later	gestaltOriginalQD and gestaltSystemVersion = $0700 or greater
Color QuickDraw class	Pre-7.0, no INITs	gestalt8BitQD
	6.0.3 or 6.0.4, and 32-Bit QuickDraw INIT v. 1.0	gestalt32BitQD
	System 6 from 6.0.5 on, and 32-Bit QuickDraw INIT v. 1.2	gestalt32BitQD12
	7.0 and later	gestalt32BitQD13
ci class	6.0.4	gestalt32BitQD + 1
	System 6 from 6.0.5 on	gestalt32BitQD12
	7.0 and later	gestalt32BitQD13

USE THE BOTTLENECKS

QuickDraw routines are easily customizable, which can be incredibly useful; however, this feature is typically underused. (In fact, most of the Macintosh is customizable. There ought to be a whole chapter in *Inside Macintosh* on customization; there are so many places in the OS that you can intercept, you could probably patch out the whole OS if you were so inclined.) You can replace QuickDraw's "guts" with viscera of your own design, completely (and reversibly) transforming QuickDraw's functionality.

Here's one example of how this can be useful: Let's say we want to find out the exact colors used in a picture that contains innumerable colors. We'll be drawing the picture to an 8-bit color monitor, and we want to manually select the best 256 colors, replacing the default color table that DrawPicture uses. There are two methods of getting the colors used in a PICT: use the System 7 Picture Utilities Package, or do it yourself. The Picture Utilities Package is available only in System 7, so if we want to run on earlier systems, our only choice is to do it ourselves. We do that by using the bottlenecks.

You can replace all the bottlenecks with no-ops except for a few carefully selected ones, then draw the picture. Your replacement bottlenecks will be able to watch all the picture data go by and can keep track, say, of the colors used in the picture. (Two sample programs on the CD, CollectPictColors and GMundo, demonstrate this technique.) So, for instance, to draw our many-colored picture with a custom-picked set of 256 colors, we actually have to draw the picture twice: the first time, we replace the bottlenecks, allowing us to use — collect, extract, or read — the colors in the picture. We can then set up the destination cGrafPort with the colors we want to show, restore the bottlenecks, and draw the picture again, this time to actually image it into the destination cGrafPort.

THAT'S IT FOR NOW

When you're faced with a question about QuickDraw, try running through the answers in this column first, to see if any of them fit. Is the state of the machine at the time of the call different than you assumed? Did you check the QuickDraw features and version? Can you do it off-screen? Can you intercept processing at the bottleneck level to customize QuickDraw's routines? It's likely that one of these answers will help.

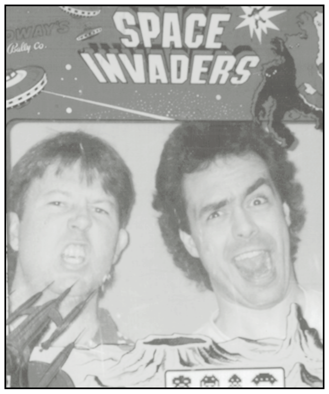
RELATED READING

- "Graphical Truffles: Animation at a Glance" by Edgar Lee, *develop* Issue 12.
- "Drawing in GWorlds for Speed and Versatility" by Konstantin Othmer and Mike Reed, *develop* Issue 10.
- "QuickDraw's CopyBits Procedure: Better Than Ever in System 7.0" by Konstantin Othmer, *develop* Issue 6.
- "Realistic Color for Real-World Applications" by Bruce Leak, *develop* Issue 1.
- *Inside Macintosh* Volume VI (Addison-Wesley, 1991), Chapter 17, "Color QuickDraw."
- *Inside Macintosh* Volume V (Addison-Wesley, 1988), Chapter 4, "Color QuickDraw."
- *Inside Macintosh* Volume IV (Addison-Wesley, 1986), Chapter 4, "QuickDraw."
- *Inside Macintosh* Volume I (Addison-Wesley, 1985), Chapter 6, "QuickDraw."
- Macintosh (Imaging) Technical Notes "Principia Offscreen Graphics Environments" (formerly #120) and "32-Bit QuickDraw: Version 1.2 Features" (formerly #275).

Thanks to Edgar Lee, Konstantin Othmer, Brigham Stevens, Forrest Tanaka, and John Wang for reviewing this column.•

VIDEO DIGITIZING UNDER QUICKTIME

With the introduction of the 'vdig' component in QuickTime 1.0, Apple established an API that encompassed the critical features of video digitizing hardware. This article takes a closer look at the more complex aspects of the QuickTime 1.0 interface and at the new functionality provided by QuickTime 1.5. Thanks to the QuickTime framework, application writers and video digitizer developers have begun to deliver the kind of high-performance solutions we've all been waiting for. And the evolution of this technology has just begun.



CASEY KING AND GARY WOODCOCK

Video digitizing hardware on the Macintosh is not new. Way back in 1990, dozens of hardware developers were offering video digitizing cards that they hoped would score big in that elusive but potentially lucrative market called multimedia. However, because these products targeted different niches and had different feature sets, there was a lot of chaos and redundancy in the marketplace. Each video digitizer card had its own API. A multimedia software developer who wanted to support more than one specific card had to either write a single application that included code for the API of every card on the market or else release multiple versions of an application, one version for each card. The inefficiency of this situation kept developers from introducing innovative products quickly. Users, for their part, were confused by the vast differences in the user interfaces of various products and in the capabilities of the hardware.

The introduction of QuickTime in December 1991 changed all this by providing a standard video digitizing interface. The component nature of QuickTime allowed video digitizer manufacturers to concentrate on making value-added hardware and software, secure in the knowledge that their products would work with whatever general-purpose video capture and editing applications were out there. Application writers could at last code to a standard interface and take advantage of improvements in the underlying hardware as they came along. Customers also benefited from a standard, and usually simplified, interaction with these devices.

CASEY KING AND GARY WOODCOCK are considered fictitious by Apple Computer, Inc. (at least, they're never around their offices when anybody's looking for them). Any similarity to actual persons, living, dead, or somewhere in between, is unintentional, unlikely, and if true, probably unfortunate. All persons appearing in this article are over 18 years of age (physically, anyway). This article was written entirely on location in Austin, TX, and Cupertino, CA, usually between the hours of 10 P.M. and 4 A.M.•

Casey King and Gary Woodcock are trademarks of Apple Computer, Inc.

So much for history! This article takes a look at the present and future of QuickTime video digitizing from two perspectives — that of the digitizer developer and that of the video application developer (although the video neophyte will find valuable information here as well). The article focuses on the less understood areas of the 'vdig' interface and the new features in QuickTime 1.5. Because a discussion of video digitizing under QuickTime would be incomplete without a look at the video digitizer's main client, the sequence grabber, we also briefly examine this powerful component. We conclude with a wish list of features for the next generation of video digitizing products.

To get the most out of this article, you need to be familiar with QuickTime components in general, video digitizer components specifically, and some basic video terminology. For an overview of components, read our article "Techniques for Writing and Debugging Components" in Issue 12 of *develop*. We also suggest that you read Chapter 7, "Video Digitizer Components," in *Inside Macintosh: QuickTime Components* (which is included in the QuickTime Developer's Kit v.1.5). Finally, while we've attempted to interject definitions of some basic video terms, the References box at the end of this article lists some of our favorite books on video. If you're really interested in dazzling your friends with your new-found video expertise, you'll want to investigate some of these books.

This issue's CD contains two pieces of sample code related to this article. The 'vdig' code is an example of a software-only digitizer. You can use it as a template to write your own video digitizer components or as a vehicle for testing grab applications when you don't have any video digitizing hardware. The sample application, HackTV, shows how to use the sequence grabber to preview and record movies. HackTV can use either the software 'vdig' provided or a hardware 'vdig' that you may already own. HackTV can also be used by 'vdig' makers to test their code.

THE VIDEO CAPTURE PROCESS

As Figure 1 shows, the process of making movies involves several components. The sequence grabber component (component type 'barg') plays an especially critical role. It's responsible for coordinating the activities of the lower-level components to achieve different results — like displaying video in a window, grabbing a single picture, or grabbing a movie. By protecting application developers from having to deal with the low-level management of the video digitizer, the sequence grabber makes it much easier to incorporate video input capabilities in applications. Note that the sequence grabber also handles audio input devices and synchronization of picture and sound. For the sake of simplicity, these tasks aren't shown in Figure 1.

Data flow in the video digitizing pipeline begins with an analog video source like a video camera, but it could also be a VCR, a laser disc, or a direct broadcast feed. The video digitizer's primary purpose in life is to convert the analog signal into a digital form that can be either displayed in the frame buffer or processed further by an

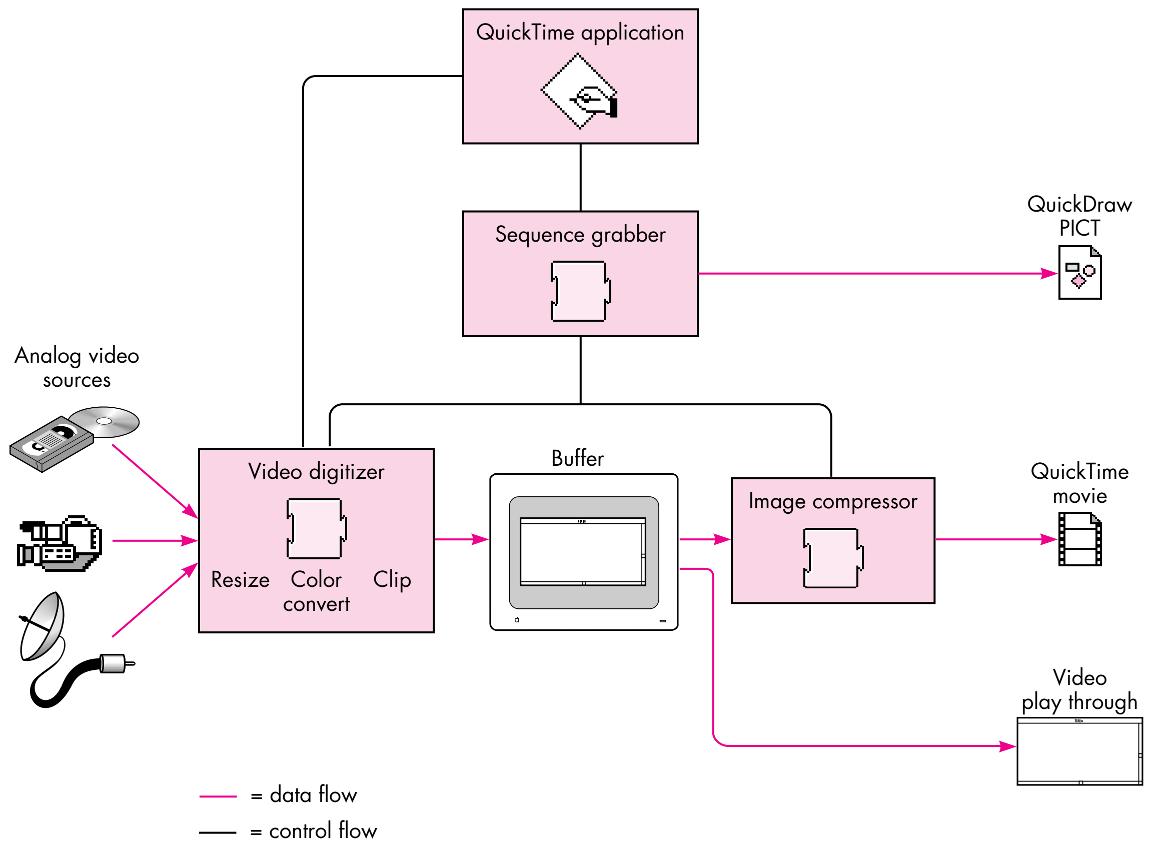


Figure 1
Grabbing Video — The Big Picture

image compressor. The video digitizing hardware can optionally perform resizing, color conversion, or clipping. In the absence of hardware support for these operations, the sequence grabber will sometimes provide them, as we'll explain later. If the application's request is for video in a window, the process is complete. (From a video digitizing perspective, displaying live video in a window is called *play through*, and capturing video to a movie is called *capturing* or *grabbing*. The sequence grabber calls these operations *previewing* and *recording*, respectively.) When a movie is requested, an image compressor processes the data further, and the result is stored either in system memory or to disk.

An important restriction of the QuickTime 1.0 video capture process is that the digitizer is limited by the image compressor. Since the grabbed image has to be perfectly still while the compressor is working, the digitizer can't grab the next frame until the compressor is finished with the current one; otherwise, there will be frame tears. Thus, the speed of the compressor has a significant influence on the effective capture rate of the digitizer. (For a discussion of industry-standard frame rates and acceptable QuickTime capture rates, see “Frame Rates and Motion Quality.”) One of the biggest challenges in making a movie is figuring out how to compress the data fast enough to maintain a good effective capture rate.

A CLOSER LOOK AT SOME 'VDIG' BASICS

In this section, we explore a few QuickTime 'vdig' topics that seem to give developers the most trouble when they first undertake the task of writing a video digitizer component. While *Inside Macintosh* is the definitive reference source for all the information we present here and for all of QuickTime, this section will give you additional insight into the more difficult aspects of rolling your own 'vdig'.

To illustrate some specific points, this section presents code excerpts from our software implementation of a 'vdig'. In practice, how you write a video digitizer component depends heavily on your particular digitizer hardware implementation. For the sake of illustration, however, we’ll simulate some hardware features with software that provides roughly equivalent functionality.

GETTING VIDEO COORDINATE SYSTEMS STRAIGHT

The coordinate system for video digitizers can seem confusing, but it’s actually quite straightforward. The common mistake is to try to map the digitizer coordinate

FRAME RATES AND MOTION QUALITY

The term *frame rate* is frequently tossed about in discussions of the pros and cons of video digitizers. Frame rate is the rate at which frames appear during video playback.

It’s commonly held that the frame rate corresponding to full-motion video is 30 frames per second (fps). However, 30 fps is not the only interesting frame rate or even the only “true” full-motion frame rate.

- 30 fps is the normal frame rate for NTSC video and broadcast production. The precise rate is actually 29.97 fps. When a QuickTime product promises “full-motion” video, this is the rate that’s typically implied.
- 25 fps is the normal frame rate for PAL and SECAM video and broadcast production.
- 24 fps is the normal frame rate for theatrical film production.
- 10 to 12 fps is widely regarded as the minimum acceptable frame rate for a QuickTime movie. At rates below this threshold, the motion is generally perceived to be too jerky.

In addition to frame rate, there are two other terms you should know about. If the frame rate measures the speed at which the movie is played back for the viewer, the *capture rate* is the rate at which the 'vdig' hardware is capable of capturing frames. The *effective capture rate* is the number of frames per second that end up in a QuickTime movie. Many factors can make the effective capture rate less than the intrinsic rate the hardware can support. We’ll discuss these factors later in the article.

Great QuickTime movies are tough to produce due to the number of tradeoffs that need to be considered. See the article “Making Better QuickTime Movies” in this issue for more information on managing frame rate, frame size, compression, image quality, and sound to achieve the best results.•

system onto the QuickDraw global coordinate system, even though the two aren't related. The critical point to keep in mind is that when referencing the video source, you're working in a coordinate system that's specific to your digitizing hardware. All cropping rectangles are relative to this coordinate system.

Figure 2 shows the four rectangles that define the video source. The MaxSrcRect defines the maximum source area that the digitizer is capable of grabbing. Typically this area includes all or portions of the vertical and horizontal blanking areas. Note that you don't have to define the top left point of MaxSrcRect as 0,0; this is an entirely arbitrary reference point that 'vdig' developers can define as they choose. The other three rectangles are defined in relation to MaxSrcRect. The ActiveSrcRect is the region of the maximum source rectangle that contains the actual video image. The first pixel of active video is the top left corner, and the last pixel is the bottom right.

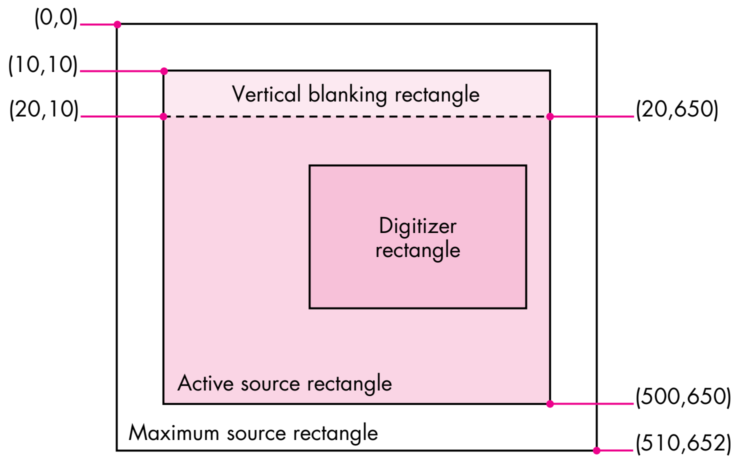


Figure 2
A Video Digitizer Coordinate System

The DigitizerRect describes the area of the MaxSrcRect to be captured — the image that the user will actually see, although it hasn't been scaled yet. The DigitizerRect defaults to the area of the ActiveSrcRect; to describe a cropped image, the DigitizerRect is usually defined as a portion of the ActiveSrcRect. It's not uncommon for part of the blanking signal to be displayed in the ActiveSrcRect. This is because different source devices — like VCRs, laser discs, and broadcast signals — send out slightly different analog signals. To align the image, a 'vdig' client can nudge the DigitizerRect a few pixels in the appropriate direction using VDSetDigitizerRect.

The last rectangle, VBlankRect, defines the area of vertical blanking. This region can contain vertical interval time code (VITC), closed captioning, and teletext. For those video dweebs out there, this corresponds to lines 10 to 19 of each field of the incoming video.

Remember, all rectangle coordinates are relative to MaxSrcRect. MaxSrcRect, ActiveSrcRect, and VBlankRect are always fixed and hardware dependent. The only control that a client has is in the definition of DigitizerRect. The following code shows how to implement the VDGetMaxSrcRect call for a 'vdig'. Implementing the VDGetActiveSrcRect and VDGetVBlankRect calls is very similar.

```
pascal VideoDigitizerError GetMaxSrcRect(Handle storage, short inputStd,
                                          Rect *maxSrcRect)
{
   long     error = noErr;

   if (inputStd == ntscIn)    // Example supports only NTSC.
      SetRect(maxSrcRect, 0, 0, kMaxHorNTSCIn, kMaxVerNTSCIn+kVerBlank);
   else
      error = paramErr;
   if (!error)
      (**storage).maxSrcRect = *maxSrcRect;
   return (error);
}
```

SetDigitizerRect, shown below, is also very straightforward. Notice that the DigitizerRect must fully intersect the MaxSrcRect.

```
pascal VideoDigitizerError vdigSetDigitizerRect(vdigGlobals storage,
                           Rect *digiRect)
{
   Rect  tempR;

   // Can't be empty.
   if (!digiRect || EmptyRect(digiRect)) return (paramErr);
   // They must intersect . . .
   if (!SectRect(digiRect, &(**storage).maxSrcRect, &tempR))
      return (paramErr);
   // . . . completely.
   if (!EqualRect(digiRect, &tempR)) return (paramErr);
   (**storage).digiRect = *digiRect;
   . . .
   // Insert hardware-dependent code to crop.
   . . .
   return noErr;
}
```

A single full-sized video frame consists of two fields, one containing the odd-numbered scan lines and the other containing the even-numbered scan lines.•

One more very important point to understand about video coordinate systems is that scaling and translation can be specified either as a matrix or as a destination rectangle. Your digitizer must support both transformation methods. If the matrix is nil, use the destination rectangle; otherwise, ignore it and use the matrix. The 'vdig' must offset the top left of the DigitizerRect to 0,0 before applying the matrix, or the video won't be positioned correctly. This isn't necessary when using the destination rectangle. By the way, the sequence grabber uses the destination rectangle, and not the matrix, to specify scaling and translation, although this may change at any time.

ACCELERATING THE FRAME CAPTURE PROCESS

Video digitizer clients — such as an application or, more likely, the sequence grabber — that want to configure a 'vdig' for video play through can do so by specifying a video source rectangle with VDSetDigitizerRect and a destination with VDSetPlayThruDestination. The live video stream is then enabled via the VDSetPlayThruOnOff call. Clients that want to capture and store images to make a QuickTime movie, on the other hand, must deal with the tradeoff between compression time and capture rate described earlier. This section explains how a video digitizer client and a 'vdig' can cooperate to maximize the effective capture rate and thus the overall performance of a digitizing system.

At the most basic level, the frame capture process involves grabbing a frame, compressing it, appending the frame to a movie, grabbing another frame, compressing it, and so on. Unfortunately, while this synchronous frame grabbing (with the VDGrabOneFrame call) is fine for single grabs, it's too slow for capturing moving images. The real killer when using the synchronous call is that no parallel processing can occur. Once the call to the video digitizer is made, the entire system is brought to a stop waiting for the completion of the frame grab. Since the digitizer must synchronize to the incoming video's vertical blanking interval, this can mean waiting the duration of up to one or two video fields (33,333 to 66,666 microseconds for NTSC video). What's more, because the video stream must be stopped at the conclusion of the call and restarted to grab another frame, the actual wait between synchronous grabs tends to be even longer.

A video digitizer client can achieve parallel processing, and thus better performance, if it grabs asynchronously, sending the image either to a single buffer or to a series of buffers. If you use just one buffer, you get a slight performance improvement over a synchronous grab. The VDGrabOneFrameAsync call tells the digitizer to kick off a frame grab and to return immediately to the caller (this typically takes around 500 microseconds). The compressor can be turned loose on the frame being grabbed without waiting for the grab to be complete. (Such *beam chasing* assumes that the compressor is slower than the video digitizer and won't catch it. Most software compressors today are in fact slower than digitizers.) Once the compressor is finished with the frame, the client makes the VDDone call to determine whether the first frame grab is complete before requesting another one. This process continues until recording is terminated.

For an example of how to set up destination characteristics, see the description of VDSetPlayThruDestination in *Inside Macintosh: QuickTime Components.*•

VDSetPlayThruDestination is one of the required calls defined in the video digitizer component API. For a complete list of all the calls that a digitizer component must implement, see the section "Required Functions" in Chapter 7 of *Inside Macintosh: QuickTime Components.*•

One problem with such single-buffer asynchronous grabs is that the client must still start and stop the video stream with each call, incurring the same overhead as in a synchronous grab. An even more serious problem is that the 'vdig' client must handle frame synchronization between the digitizer and compressor; otherwise, the resulting movie will contain annoying frame tears. Why does this happen? Suppose the VDGrabOneFrameAsync call is made while the digitizer is in the middle of a frame. The digitizer will honor the request and return immediately, but the video won't really start until the beginning of the next field. If the compressor is turned loose as soon as the digitizer returns, it will be compressing stale data. And depending on how slow the compressor is, new digitizer data may overwrite the stale data in midstream, all of which is bad news.

Clients of your 'vdig' can attain still better performance if the 'vdig' implements multiple buffers. By constantly ping-ponging between two or more buffers, the 'vdig' can increase the effective capture rate, potentially to the maximum rate. The process works roughly as follows: A 'vdig' client initializes the process by making the video digitizer call VDSetupBuffers (we'll look at this in more detail momentarily). The client then begins the capture process by calling VDGrabOneFrameAsync to fill buffer 1 with a frame and to start the next grab into buffer 2. The client calls VDDone to make sure that the frame grab in buffer 1 is complete. Next, the compressor compresses the frame in buffer 1. Then VDGrabOneFrameAsync is called to send the next digitized frame back into buffer 1, and the compressor starts compressing the contents of buffer 2. The whole sequence is repeated for successive frames.

If you think two buffers are a snap, that's great, because in practice there are often three. Figure 3 shows a snapshot of the asynchronous grab process when three buffers are implemented. The number of buffers your 'vdig' needs to implement, by the way, is typically equal to the number of concurrent operations a video digitizing system must perform on those buffers. The example shown in Figure 3 uses three buffers to support *interframe compression.* This technique, which is also known as *frame differencing*, makes it possible for a compressor to eliminate the redundant data in a sequence of frames — essentially the parts of the image that stay the same from one frame to the next. At specified intervals, all the data in a frame is saved; this frame is known as a *key frame*. Depending on the content of the image sequence, frame differencing can provide a substantial increase in compression efficiency.

Figure 3 shows what's going on in three hypothetical buffers at a specific moment in time. The whole process of grabbing a frame from the digitizer and compressing it with frame differencing involves the following steps:

1. The 'vdig' initially uses a buffer — in this case, buffer 1 — as a play-through destination.
2. The sequence grabber makes the VDGrabOneFrameAsync call with the next available buffer, buffer 2.

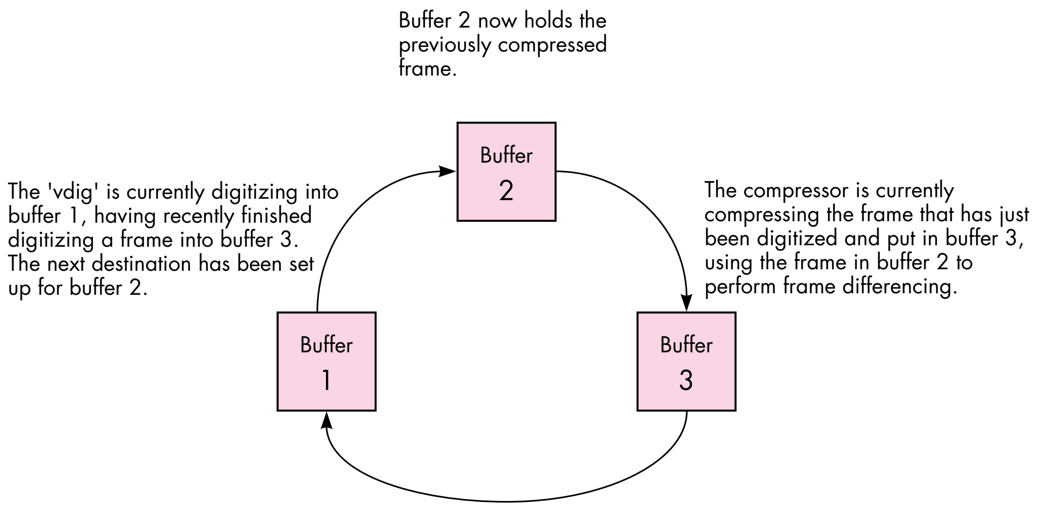
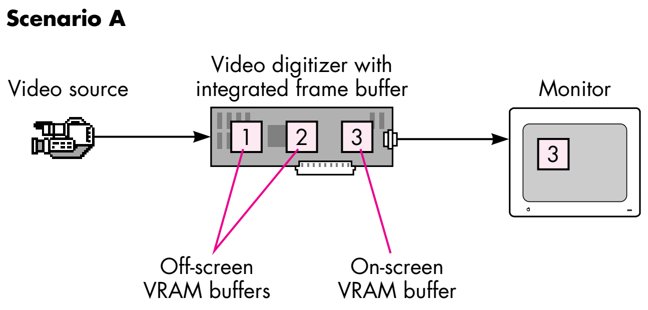

Buffer 2 now holds the previously compressed frame.
The 'vdig' is currently digitizing into buffer 1, having recently finished digitizing a frame into buffer 3. The next destination has been set up for buffer 2.
The compressor is currently compressing the frame that has just been digitized and put in buffer 3, using the frame in buffer 2 to perform frame differencing.
Buffer 1
Buffer 2
Buffer 3

Figure 3
Multiple Buffers in Motion

3. The 'vdig' receives VDGrabOneFrameAsync, returns to the sequence grabber, and begins the asynchronous grab — in this case, completing a frame grab into buffer 1. Upon completion it's available to begin digitizing into buffer 2.
4. The sequence grabber uses VDDone to find out when the frame grab into buffer 1 is complete.
5. The sequence grabber makes another VDGrabOneFrameAsync call. The 'vdig' completes the frame grab into buffer 2 and then is available to begin digitizing into buffer 3.
6. The sequence grabber calls the Image Compression Manager to compress the completed frame, while the 'vdig' continues digitizing.
7. The sequence grabber writes the compressed frame to disk.
8. The sequence grabber repeats the basic process outlined in steps 4 through 7, this time using buffer 3 for the next buffer, compressing buffer 2, and using buffer 1 as the first key frame for frame differencing.
9. The sequence grabber repeats the process outlined in steps 4 through 7, using buffer 1 for the next buffer, compressing buffer 3, and using buffer 2 as the next key frame for frame differencing.
10. The 'vdig' is returned to play-through mode when the record operation is completed. A movie file is created.

A word of caution: A common mistake in using VDGrabOneFrameAsync is to pass in the current buffer to be worked on rather than the next destination buffer. By telling the digitizer in advance where the next frame should be placed, you give the digitizer all the information it needs to do a fast buffer change once the current frame grab is complete. The time required to reposition the video buffer is less than the time in a vertical blanking period, so the digitizer theoretically has enough time to grab every frame without the need for resynchronization. In this case, the obstacles to achieving the maximum capture rate are the other processes involved, such as compression and disk access speeds. Note that there's no need to turn the digitizer on and off with multiple buffers because the video frame in buffer 1 will in effect be frozen once the digitizer repositions its free-running output to buffer 2.

Now let's take a step back and look at the interaction between the video digitizer client and the 'vdig' during buffer initialization. Figure 4 shows two possible scenarios. In scenario A, the card has local memory available for multiple buffering but doesn't support generic hardware DMA. (Digitizers that support hardware DMA



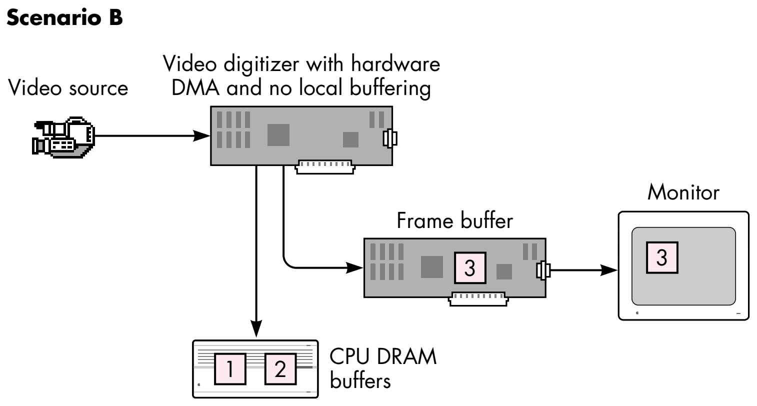


Figure 4
Initializing Multiple Buffers — Two Scenarios

can send the video data to any available memory; they aren't restricted to local memory.) In scenario B, the digitizer supports hardware DMA but doesn't have local memory for multiple buffering.

For both scenarios in Figure 4, the basic sequence of events is similar.

1. The sequence grabber uses VDSetPlayThruDestination to set up the first destination, which happens to be visible to the user on a monitor.
2. The sequence grabber determines how many buffers to allocate based on the video characteristics, the compression settings, and the amount of memory available for multiple buffering.
3. The sequence grabber uses VDSetupBuffers to tell the video digitizer how to partition the buffers.

The differences between the scenarios depicted in Figure 4 are subtle but very important to understand. First, in scenario A, the sequence grabber uses the integrated frame buffer when setting up the play-through destination. In scenario B, where the digitizer is more flexible in its ability to redirect video data, the sequence grabber can make the play-through destination any frame buffer — that is, any screen — that the user chooses.

The second difference between the two scenarios in Figure 4 is in how the sequence grabber finds available memory and partitions that memory into multiple buffers. Because the digitizer in scenario A doesn't support hardware DMA, local memory on the card is the only memory available for multiple buffering. The sequence grabber will make the VDGetMaxAuxBuffer call to determine the maximum usable memory available. In our example, three buffers are allocated. If sufficient space for three buffers isn't available, the sequence grabber will make the buffer sizes smaller and use software to expand the frame to the desired size. In contrast, the video digitizer in scenario B supports hardware DMA and doesn't have any local buffering, so the sequence grabber will use system memory for buffer allocation. In this case the size of the buffers is limited by the amount of free memory available. Once the number and size of the buffers have been determined, VDSetupBuffers is used in both cases to communicate this information to the 'vdig'.

To sum up, a 'vdig' client looks at two video digitizer capabilities to determine how to initialize buffers — the availability of local off-screen memory and the digitizer's ability to do generic hardware DMA. While there are actually four possible combinations of these two capabilities, the two scenarios shown in Figure 4 define the major options for initializing multiple buffers. The two combinations not shown in the figure are a 'vdig' that doesn't provide any local off-screen memory and doesn't support hardware DMA, and a 'vdig' that supports both capabilities. If a 'vdig' has neither capability, it's not a very interesting digitizer — just consider yourself lucky that it can digitize video into a window at all. At the other end of the spectrum, the increasing numbers of digitizers that support both local memory and hardware DMA are *very* interesting. With these digitizers, buffers are allocated in the local off-screen memory first for performance reasons, while hardware DMA is normally used to display video in a window on any screen.

At this point you're probably asking yourself, "Do I really need to understand this stuff?" The answer for video digitizer developers is a resounding "Yes," especially if you want to squeeze out every last drop of performance. For those interested only in the general concepts, we've probably led you down a road you wish you hadn't embarked on. Sometimes the quest for new knowledge hurts!

If you're writing a digitizer, you'll be interested in the following code, which shows how to write the routines that support multiple-buffer asynchronous frame grabs. Notice that in the vdigSetupBuffers routine, pendingAsyncBuffer is used to keep track of the next buffer to be grabbed. Here it's initialized to -1, which is not a valid buffer number.

```
pascal VideoDigitizerError vdigSetupBuffers(vdigGlobals storage,
                                       VdigBufferRecListHandle bufferList)
{
   OSErr          err;
   MatrixRecord   matrix;
   short          i;
   RgnHandle      clipRgn;

   if (!bufferList) return paramErr;           // Can't have empty list.
   if (!(**bufferList).count) return paramErr;  // Can't have 0 buffers.
   // Dispose of any buffers previously created.
   if ((**storage).bufferList) {
      DisposeHandle((Handle)(**storage).bufferList);
      (**storage).bufferList = 0;
   }
   // Same with any clipRgn previously created.
   if ((**storage).clipRgn) {
      DisposeRgn((**storage).clipRgn);
      (**storage).clipRgn = 0;
   }
   // Don't accept a mask if the 'vdig' can't clip.
   if (!gCanClip && (**bufferList).mask) return paramErr;
   // Copy the matrix if it exists.
   if ((**bufferList).matrix)
      matrix = *(**bufferList).matrix;
   // Make a local copy.
   HandToHand((Handle *)&bufferList);
   if (err = MemError()) return err;
   if (clipRgn = (**bufferList).mask) {
      HandToHand((Handle *)&clipRgn);
      if (err = MemError()) return err;
      (**storage).clipOrigin = (**bufferList).list[0].location;
   }
   // Save the important stuff in private storage for later retrieval.
   (**storage).bufferList = bufferList;
   (**storage).clipRgn = clipRgn;
   (**storage).pendingAsyncBuffer = -1;
   (**storage).matrix = matrix;
   // Do the important error checking when it's not performance-critical.
   for (i=0; i < (**bufferList).count; i++) {
      if (!validatePixMap(storage, (**bufferList).list[i].dest))
         return paramErr;
   }
   return noErr;
}
```

The vdigGrabOneFrameAsync routine (below) is continually being called by the sequence grabber once movie making starts. After some simple error checking, the software digitizer draws the frame at the appropriate destination. The buffer used in drawVideoFrame is the pending buffer that was passed into this routine on the last call. While this isn't so important with the software-only video digitizer illustrated here, it can give hardware digitizers some additional time to get things switched over and started in anticipation of the next vdigGrabOneFrameAsync call.

```
pascal VideoDigitizerError vdigGrabOneFrameAsync(vdigGlobals storage,
                                                  short buffer)
{
   VdigBufferRecListHandle bufferList;
   // Bail if you can't do asynchronous grabs.
   if (!gCanAsync) return digiUnimpErr;
   // Make sure the buffer list is set up first with VDSetupBuffers.
   if (!(bufferList = (**storage).bufferList)) return badCallOrder;
   // Make sure the buffer request is within bounds.
   if (buffer > (**bufferList).count) return paramErr;
   // Get the buffer to draw into from the last one saved.
   // This is the one saved in pendingAsyncBuffer.
   if ((**storage).pendingAsyncBuffer != -1) {
      short aBuf = (**storage).pendingAsyncBuffer;
      if (aBuf == buffer)
         DebugStr("\pasync grab into incomplete buffer");
      drawVideoFrame(storage, (**bufferList).list[aBuf].location,
         (**bufferList).list[aBuf].dest);
   }
   // Set up the next buffer to use when called.
   (**storage).pendingAsyncBuffer = buffer;
   return noErr;
}
```

IDENTIFYING DIGITIZER TYPES AND CAPABILITIES

No two video digitizers are alike. To make sure your 'vdig' component works smoothly with QuickTime, it's critical to identify the capabilities your hardware provides.

The 'vdig' component interface attempts to be very flexible in allowing you to indicate what your card can do. A 'vdig' specifies its type and capabilities in the DigitizerInfo structure, shown below. Two calls — VDGetDigitizerInfo and VDGetCurrentFlags — give a client (normally the sequence grabber) access to information contained in this structure.

```
typedef struct {
   short       vdigType;
   long        inputCapabilityFlags;
   long        outputCapabilityFlags;
   long        inputCurrentFlags;
   long        outputCurrentFlags;
   short       slot;
   GDHandle    gdh;              // For vdigs with preferred screen
   GDHandle    maskgdh;          // For vdigs that have mask planes
   short       minDestHeight;    // Smallest resizable height
   short       minDestWidth;     // Smallest resizable width
   short       maxDestHeight;    // Largest resizable height
   short       maxDestWidth;     // Largest resizable height
   short       blendLevels;      // # of blend levels = 2 if 1-bit mask
   long        reserved;
} DigitizerInfo;
```

In the vdigType field, you specify which of the four types of digitizer you are. Either you're a basic rectangular digitizing device or you're a device that supports clipping — an alpha channel device, a mask plane device, or a key color device. In the capability flags fields, you indicate to clients what capabilities a particular digitizer instance provides.

The current flags fields have the same attribute bit fields as the capability flags, but they indicate the currently available capabilities, not the total possible capabilities. By nature, some capabilities are mutually exclusive. For instance, if you support NTSC and PAL input formats, at any given time you're actively doing only one of them. The bit corresponding to the active standard is the one that would be set in the current flags, while both would be set in the capability flags.

Figure 5 lists each of the attribute flags for input and output capabilities, with the flags added to the API in QuickTime 1.5 shown in bold. Complete descriptions of the flags can be found in Chapter 7, "Video Digitizer Components," of *Inside Macintosh: QuickTime Components*.

One point we'd like to emphasize is that clients of your digitizer will be much happier if you truthfully state what you can and can't do. The sequence grabber, in particular, will function much better. So don't broadcast that you can support hardware play through or hardware DMA unless you can! In addition, when designing your device's feature set, it's better not to make your capabilities modal. For instance, don't build a card that can resize in 32-bit-per-pixel mode but not in 16-bpp mode.

Applications also need to know about your capabilities. An application makes preflight calls, like VDPreflightDestination, to your digitizer to determine whether its request will be honored, denied, or changed. Digitizers are required to support all the preflight calls.

Some examples. To see how the attribute flags are set, let's take a look at three completely different fictional video digitizer implementations. One of the examples includes the support provided by QuickTime 1.5 for digitizer hardware compression — a nifty feature we'll discuss when we look at the QuickTime 1.5 additions later in this article.

As you consider the examples, pay particular attention to the use of the following structure members and attribute flags, which seem to be among the least understood:

```
vdigInfo->inputCapabilityFlags[digiInVTR_Broadcast]
vdigInfo->outputCapabilityFlags[digiOutDoesDMA]
vdigInfo->outputCapabilityFlags[digiOutDoesDouble]
vdigInfo->outputCapabilityFlags[digiOutDoesQuad]
vdigInfo->outputCapabilityFlags[digiOutDoesHWPlayThru]
vdigInfo->outputCapabilityFlags[digiOutDoesAsyncGrabs]
vdigInfo->gdh
vdigInfo->maskgdh
vdigInfo->blendLevels
```

The size of the MaxAuxBuffer is also very important if the device supports one. The only way for a client to determine this is to make the VDGetMaxAuxBuffer call to see if it returns valid data. See "What the $#%!! Is a MaxAuxBuffer, and Do I Have One?" for details.

For our first example of a fictional digitizer, let's suppose SuperOps announces an entry-level video digitizing card with the following capabilities: The card has one video input (an RCA-style connector) and can support only the NTSC video

	Output flags		Input flags
31	–	31	Signallock
30	–	30	–
29	**PlayThruDuringCompress**	29	–
28	**CompressOnly**	28	–
27	**Compress**	27	–
26	**UnreadableScreenBits**	26	–
25	AsyncGrabs	25	–
24	KeyColor	24	–
23	Inverse LUT	23	–
22	Hardware play through	22	–
21	Hardware DMA	21	–
20	Warp	20	–
19	Blend	19	–
18	Skew	18	–
17	Vertical flip	17	–
16	Horizontal flip	16	–
15	Rotation	15	–
14	Sixteenth	14	–
13	Quarter	13	Black & white
12	Quad	12	Color
11	Double	11	VTR_Broadcast
10	–	10	Component
9	Mask	9	SVideo
8	Shrink	8	Composite
7	Stretch	7	Genlock
6	Dither	6	–
5	32 bpp	5	–
4	16 bpp	4	–
3	8 bpp	3	–
2	4 bpp	2	SECAM
1	2 bpp	1	PAL
0	1 bpp	0	NTSC

– = Reserved
bold = New for QuickTime 1.5

Figure 5
Video Digitizer Attribute Flags

WHAT THE $#%!! IS A MAXAUXBUFFER, AND DO I HAVE ONE?

VDGetMaxAuxBuffer is an extremely misunderstood 'vdig' call. For a video digitizer that implements this call, the sequence grabber can provide improved capture rates and enhanced capabilities, even if the digitizer hardware wasn't originally designed with off-screen buffering in mind.

Simply put, a MaxAuxBuffer is the total unused off-screen local memory on a digitizer card. The sequence grabber can use the MaxAuxBuffer to implement resizing, clipping, and color conversion operations for digitizers that don't support these operations in hardware. The MaxAuxBuffer can also be used to provide a low-performance equivalent to DMA for devices that don't support hardware DMA. Alternatively, the sequence grabber can use the MaxAuxBuffer for multiple buffering by partitioning it into one or more smaller buffers.

Even if your card supports high-performance DMA–to–system memory operations, we suggest that video digitizers support and implement this feature whenever possible. Transferring data into your local memory system will be faster than transferring it across NuBus™ into system memory. This, in turn, makes it possible for clients to achieve a faster effective capture rate.

A couple of rules of thumb for using VDGetMaxAuxBuffer: First, because a MaxAuxBuffer is composed of local memory only from your device, you should *not* allocate system memory to support this call. Second, the sequence grabber will sometimes act like a snapperhead and make this call multiple times. Just be consistent with your replies. You aren't responsible for managing or partitioning this memory — you simply have to say it's around.

standard. The card is a combined video digitizer and frame buffer. The frame buffer can support 1, 2, 4, 8, 16, and 32 bpp on a monitor that's 640 by 480 pixels. The video digitizer, however, can display real-time video on its own frame buffer only in the 16-bpp and 32-bpp modes. Our SuperOps card doesn't support compression, but it does support clipping via an alpha channel. Because it can resize the video but not zoom, scaling only makes the video smaller. The card has additional local memory for grabbing in 16-bpp mode, but not in 32-bpp mode.

The interesting fields and their values for the SuperOps card are:

```
vdigType                = vdTypeAlpha;
inputCapabilityFlags    = digiInDoesNTSC | digiInDoesComposite |
                          digiInDoesColor;
outputCapabilityFlags   = digiOutDoes16 | digiOutDoes32 |
                          digiOutDoesShrink | digiOutDoesMask |
                          digiOutDoesQuarter | digiOutDoesSixteenth |
                          digiOutDoesBlend | digiOutDoesHWPlayThru |
                          digiOutDoesAsyncGrabs;
gdh                     = // Handle to frame buffer graphics device
blendLevels             = video16 ? 2 : (video32 ? 256:0);
```

In 32-bpp mode, there's no room for a MaxAuxBuffer, so the call to VDGetMaxAuxBuffer fails. In 16-bpp mode, a MaxAuxBuffer is available because only half of the on-board local frame buffer memory is being used, and the size of the unused memory is equivalent to the display size, 640 by 480 pixels.

The second fictional card, the OneShot-O-Matic, is designed to do only single-frame grabs. The card has three inputs — composite, S-Video, and component RGB — and can support both the NTSC and PAL video standards on all three. The maximum size that can be grabbed is a function of which input standard the card uses (for example, the maximum size would be 768 by 576 pixels when decoding the PAL standard). Because the OneShot-O-Matic can grab only to its local memory, it can't do hardware DMA — that is, it doesn't support play through to any frame buffer. The OneShot-O-Matic supports only 32-bpp grabs. Resizing, clipping, and hardware compression aren't supported. The board is populated with 2 megabytes of memory, which is enough to support the maximum size required by the PAL standard.

For the OneShot-O-Matic card, the interesting fields and their values are:

```
vdigType                =  vdTypeBasic;
inputCapabilityFlags    =  digiInDoesNTSC | digiInDoesPAL |
                           digiInDoesComposite | digiInDoesSVideo |
                           digiInDoesComponent | digiInDoesColor;
outputCapabilityFlags   =  digiOutDoes32 | digiOutDoesAsyncGrabs;
gdh                     =  nil;
blendLevels             =  0;
```

In this case, the MaxAuxBuffer will consume the entire 2 MB local buffer area. This works out to be 524,288 available pixels in 32-bpp mode (2097152 / 4 bytes per pixel). The representation of this as a width and height must be large enough to support the largest grab size; thus 800 by 655 pixels would be valid, as would 700 by 748 pixels. Hardware resizing would have allowed multiple buffers to reside in the MaxAuxBuffer. With this card, however, the video digitizer will be relying on the sequence grabber to do resizing, color conversion, and on-screen placement operations.

Our third fictional card, the Verne-Motion, is designed to perform not only video digitizing, but also hardware compression using a highly proprietary algorithm called SqueezeMe. (We'll discuss compressed-source devices shortly.) The card has one S-Video input and supports all input standards — NTSC, PAL, and SECAM. In addition, the Verne-Motion can simultaneously support two channels of video: an RGB pixel stream formatted for display, and a compressed SqueezeMe stream to be saved as a QuickTime movie. The card has a general-purpose DMA engine that can direct these data streams either to memory on the card or to system memory. The on-card memory is 4 MB and gives better performance than you get when you send the video data to system memory. The Verne-Motion supports hardware clipping with an 8-bit mask plane, which can also be stored in local memory or in system memory. The video input circuitry has a mode that distinguishes between clean broadcast-quality input signals and noisier VCR-type signals. Finally, the Verne-Motion lets you arbitrarily scale video images to make them up to two times larger in both the horizontal and vertical directions.

The interesting fields and their values for the Verne-Motion card are:

```
vdigType                = vdTypeMask;
inputCapabilityFlags    = digiInDoesNTSC | digiInDoesPAL |
                          digiInDoesSECAM | digiInDoesSVideo |
                          digiInVTR_Broadcast | digiInDoesColor;
outputCapabilityFlags   = digiOutDoes8 | digiOutDoes16 | digiOutDoes32 |
                          digiOutDoesStretch | digiOutDoesShrink |
                          digiOutDoesMask | digiOutDoesDouble |
                          digiOutDoesQuarter | digiOutDoesSixteenth |
                          digiOutDoesHW_DMA | digiOutDoesHWPlayThru |
                          digiOutDoesAsyncGrabs | digiOutCompress |
                          digiOutPlayThruDuringCompress;
gdh                     = nil;
maskgdh                 = // Handle to local mask plane – 8 bpp deep
blendLevels             = 256;
```

The size of the MaxAuxBuffer in this case would be 1024 by 1024 pixels in 32-bpp mode (1024 * 1024 * 4 bytes per pixel equals 4 MB, which is the total available memory). In the 16-bpp mode, the MaxAuxBuffer would be 2048 by 1024 pixels, and in the 8-bpp mode, it would be 2048 by 2048 pixels.

As you can see from these examples, there are many different classes of video digitizers out there. Your card may look very similar to one of the ones we've presented, but it's more likely that you'll have to set different flags to describe your card's features.

THE POWER OF THE SEQUENCE GRABBER

The sequence grabber makes life easier for application programmers by handling all the messy details of controlling video digitizers, sound input devices, and compressors. This marvelous piece of QuickTime has a very rich API that we won't even pretend to cover in this article. However, the sequence grabber plays such an important role in a video digitizing application that a brief introduction is in order. Note that while the code excerpts below demonstrate how the sequence grabber can preview and record a video channel, the HackTV application on the CD includes an audio channel as well.

Before we discuss previewing and recording, we want to briefly describe how the sequence grabber can make up for functionality missing from a digitizer. For the sequence grabber to be able to do this, the video digitizer must be able to use off-screen memory — either local memory on the digitizing device or, in the more general case, any memory. (Recall that you can use the VDGetMaxAuxBuffer call to locate local device memory.) As an example, let's see how the sequence grabber can support color conversion.

Suppose a video digitizer can display video only to its own integrated frame buffer. In a multimonitor system, this would mean a user could display video on only one of the monitors. The sequence grabber comes to the rescue because it can use off-screen memory for the grab and copy it to the desired destination buffer, making the video appear on one of the unsupported displays. Now suppose the digitizer is asked to display video in a depth it doesn't support (say the user changes the graphics mode from 32 bpp to 4 bpp with the Monitors control panel). The sequence grabber can again use off-screen memory to do the right thing. Of course, you never get something for nothing. All this buffer copying will adversely affect performance.

PREVIEWING

Previewing video with the sequence grabber is the equivalent of setting up the 'vdig' in play-through mode to display live video on the computer screen. The code to make this happen is shown below. Remember that the sequence grabber can simulate live video play through for devices that don't support hardware play through.

```
// Find and open a sequence grabber.
gSeqGrabber = OpenDefaultComponent(SeqGrabComponentType, (OSType) 0);
// If we get a sequence grabber, set it up.
if (gSeqGrabber != 0L) {
   // Get the monitor — in this case, the dialog
   // window in which video is displayed.
   gMonitor = GetNewDialog(kMonitorDLOGID, nil, (WindowPtr) -1L);
   if (gMonitor != nil) {
      // Initialize the sequence grabber.
      GetPort(&savedPort);
      SetPort(gMonitor);
      ShowWindow(gMonitor);
      result = SGInitialize(gSeqGrabber);
      if (result == noErr) {
         result = SGSetGWorld(gSeqGrabber, (CGrafPtr) gMonitor, nil);
         // Get a video channel.
         result = SGNewChannel(gSeqGrabber, VideoMediaType,
                                 &gVideoChannel);
         if ((gVideoChannel != nil) && (result == noErr)) {
               short width;
               short height;
               gQuarterSize = false;
               gHalfSize = true;
               gFullSize = false;
               result = SGGetSrcVideoBounds(gVideoChannel,
                                             &gActiveVideoRect);
               width = (gActiveVideoRect.right-gActiveVideoRect.left)/2;
               height = (gActiveVideoRect.bottom-gActiveVideoRect.top)/2;
               SizeWindow (gMonitor, width, height, false);
               // Preview the video only.
               result = SGSetChannelUsage(gVideoChannel, seqGrabPreview);
               result = SGSetChannelBounds(gVideoChannel,
                                             &(gMonitor->portRect));
         }
         // Go!
         if (result == noErr) result = SGStartPreview(gSeqGrabber);
      }
      SetPort(savedPort);
   }
}
```

We establish a connection to the sequence grabber component in the usual way, with OpenDefaultComponent. We initialize the sequence grabber, set up the graphics environment, and allocate a new channel for video. We want the displayed video to be one-quarter size, so we call SGGetSrcVideoBounds to see what size the source video is. This function calls VDGetDigitizerRect, which returns the source video size equal to the digitizer rectangle. We scale the height and width accordingly, and the new size is sent to the sequence grabber via the SGSetChannelBounds routine. Finally, we call SGStartPreview, which turns on the video digitizer by calling the digitizer function VDSetPlayThruOnOff, and previewing begins.

RECORDING

Recording is very similar to previewing and is almost as simple. The following code highlights the differences between recording and previewing.

```
// Start exactly the same way as for previewing.
// Find and open a sequence grabber.
gSeqGrabber = OpenDefaultComponent(SeqGrabComponentType, (OSType) 0);
// If we get a sequence grabber, set it up.
if (gSeqGrabber != 0L) {
   . . .
      if ((gVideoChannel != nil) && (result == noErr)) {
         // Set up size the same as in previewing case.
         short width;
         short height;
         gQuarterSize = false;
         gHalfSize = true;
         gFullSize = false;
         result = SGGetSrcVideoBounds(gVideoChannel, &gActiveVideoRect);
         width = (gActiveVideoRect.right-gActiveVideoRect.left)/2;
         height = (gActiveVideoRect.bottom-gActiveVideoRect.top)/2;
         SizeWindow(gMonitor, width, height, false);
         // Record and play images instead of just previewing them.
         result = SGSetChannelUsage(gVideoChannel, seqGrabRecord |
                                    seqGrabPlayDuringRecord);
         result = SGSetChannelBounds(gVideoChannel,
                                    &(gMonitor->portRect));
         // Get ready. . .
         result = SGSetDataOutput(gSeqGrabber,&gMovieFile,seqGrabToDisk);
         result = SGPrepare(gSeqGrabber, true, true);
         // Go!
         result = SGStartRecord(gSeqGrabber);
         while (!Button() && !result) {
            result = SGIdle(gSeqGrabber);
         }
         result = SGStop(gSeqGrabber);
      }
   . . .
}
```

There are several differences between recording and previewing. First we set up the sequence grabber to record and to play through while recording. We next specify a file for the movie to be written to, indicating that the movie be grabbed directly to disk. For a short movie, we could grab to memory if we wanted. By default, the recording time is limited by the system resources available — in this case, disk space.

The SGStartRecord call initiates the grab to disk. SGIdle is called repetitively to provide processing time to the sequence grabber. You should call SGIdle as often as possible while recording. When the user clicks the mouse button, or when the disk is full, recording will stop, and we call SGStop to complete the recording process.

That's all there is to simple recording. If you want to do more sophisticated tasks with the sequence grabber, such as replacing the standard sequence grabber disk- or compression-bottleneck routines with your own, consult *Inside Macintosh: QuickTime Components*, or refer to the SGSample sample code on the QuickTime 1.0 Developer's CD-ROM.

WHAT'S NEW FOR DIGITIZERS IN QUICKTIME 1.5

QuickTime 1.5 provides expanded support for sophisticated new types of video digitizing hardware. First, it defines a number of routines that allow digitizers to describe their capabilities to clients more completely. It also defines routines that give clients more control over the digitization process. QuickTime 1.5 adds to the video digitizer API by supporting devices that are capable of producing compressed-image data. Finally, a standard user interface has been introduced for choosing and configuring video digitizers.

EXPANDED INFORMATION SERVICES

QuickTime 1.5 encompasses a greater range of digitizer capabilities than did QuickTime 1.0. The latest API defines new flags and several new routines to allow applications to determine whether a digitizer supports these new capabilities.

New digitizer information flags. Four new flags have been defined in the outputCapabilityFlags field of the DigitizerInfo record.

- The digiOutDoesUnreadableScreenBits flag indicates that the video digitizer may put pixels on the screen that are visible but can't be used when compressing images. In other words, image data can't be read directly from the screen as source material for the compression phase.
- The digiOutDoesCompress flag indicates that the digitizer is capable of producing compressed-image data directly.
- The digiOutDoesCompressOnly flag indicates that the digitizer provides compressed-image data only and that it's unable to provide image data for display.
- The digiOutDoesPlayThruDuringCompress flag indicates that the digitizer can display image data while it's compressing it.

New information routines. Four new functions allow applications to get additional information about the digitizer's capabilities.

- The VDGetSoundInputDriver call gives video digitizers a way to tell applications which sound input driver they're associated with. Several digitizer boards now available have sound as well as video digitizing capability.
- The VDGetPreferredTimeScale routine allows digitizers that can time-stamp the data they create to communicate their preferred time scale to applications. The sequence grabber, in particular, uses this call to establish the time scale of the video data it creates from the digitizer output.
- The VDGetDataRate routine is extremely useful because it gives clients a way to determine the performance capabilities of a digitizer. The call returns three values. The first value, milliSecPerFrame, indicates the number of milliseconds of overhead involved in digitizing a single frame. *Overhead* is defined as the average delay between the time the digitizer requests a

For more information on time scales, see *Inside Macintosh: QuickTime Components* and "Time Bases: The Heartbeat of QuickTime" in *develop* Issue 12.•

frame from its associated hardware and the time the device actually delivers the frame. The second value, framesPerSecond, is the maximum rate at which the digitizer can capture video frames in its current configuration. This value is not affected by the VDSetFrameRate call, described later on. The last value, bytesPerSecond, indicates the data rate at which a compressed-source digitizer can produce compressed-image data. This value varies depending on parameters for spatial and motion quality, image size, and depth. In other words, unlike milliSecPerFrame and framesPerSecond, bytesPerSecond *isn't* a static value.

- The VDGetDMADepths routine allows an application to determine the pixel depths a DMA-style digitizer can support. The depthArray parameter is a pointer to a long word of flags, with each flag representing a possible depth. A flag with a value of 1 indicates that the corresponding depth is supported. The preferredDepth parameter is a pointer to a long word indicating — big surprise — the preferred depth of the digitizer. A value of 0 indicates that all depths are equally acceptable. Note that if a DMA-style digitizer doesn't support this call, the digitizer is assumed to be capable of handling off-screen buffers at all depths indicated in the outputCapabilityFlags field of its DigitizerInfo record.

ENHANCED DIGITIZATION CONTROL

Two new routines in QuickTime 1.5 give applications greater control over how a video digitizer performs its task. Video digitizers that can time-stamp the video frames they produce should implement the new VDSetTimeBase routine so that applications can specify the time coordinate system the video digitizer should use when time-stamping video frames.

The second routine, VDSetFrameRate, allows applications to tell a digitizer the precise frame rate to use for capture. Digitizers used to capture video at only one frame rate — as fast as possible. However, the advent of full-frame-rate digitizing hardware and compressed-source devices has made it increasingly important for clients to manage the tradeoff between frame rate, image size, and compression quality. The rate in VDSetFrameRate is expressed as a fixed-point value, typically between 0 and 29.97 (see "Frame Rates and Motion Quality" earlier in this article for a discussion of frame rate values).

COMPRESSED-SOURCE DEVICE SUPPORT

OK, pay attention here — this enhancement alone is worth the price of admission for QuickTime 1.5. QuickTime is finally poised to silence all complaints about "postage stamp video sizes" and "jerky frame rates." By expanding the video digitizer API to support compressed-source video digitizers, QuickTime gives users access to full-size, full-frame-rate digital video. A number of new video capture boards with on-board compression and decompression capabilities are already shipping, and there will undoubtedly be others soon.

QuickTime 1.5 includes eight new routines for servicing compressed-source video digitizers. Several of these calls should look familiar to you, since they're largely compressed-source versions of existing calls and serve very similar purposes.

- VDGetCompressionTypes is a straightforward call. It simply returns a handle to the list of compressors that your video digitizer implements. Each element of the list contains the component ID, type, name, format, and capabilities of a compressor.
- The VDSetCompression routine specifies which of all the possible compressors a digitizer should use. The parameters for VDSetCompression specify the spatial quality, temporal quality, depth, and other characteristics of the compression.
- VDCompressOneFrameAsync starts the digitizing process for compressed-source devices, just as VDGrabOneFrameAsync starts the process for regular digitizers. The major difference is that a compressed-source digitizer handles all the management of data buffers itself, without external assistance from the caller.
- VDCompressDone is similar to VDDone in that it allows a caller to determine when a frame has been completed (in this case, digitized and compressed).
- The VDReleaseCompressBuffer tells a compressed-source device to free the buffer returned by the VDCompressDone call.
- The VDResetCompressSequence call instructs a digitizer to insert a key frame into a frame-differenced image sequence as soon as possible after it receives this call.
- The VDGetImageDescription routine prompts the digitizer to return an image description structure corresponding to the current settings. This structure is defined in the Image Compression Manager chapter of *Inside Macintosh: QuickTime* and is the same structure that's used to describe image data in movie files.
- The VDSetCompressionOnOff routine starts and stops the digitizer. To give the digitizer adequate time to prepare itself for the requested operation, clients must call this routine before calling VDSetCompression or VDCompressOneFrameAsync.

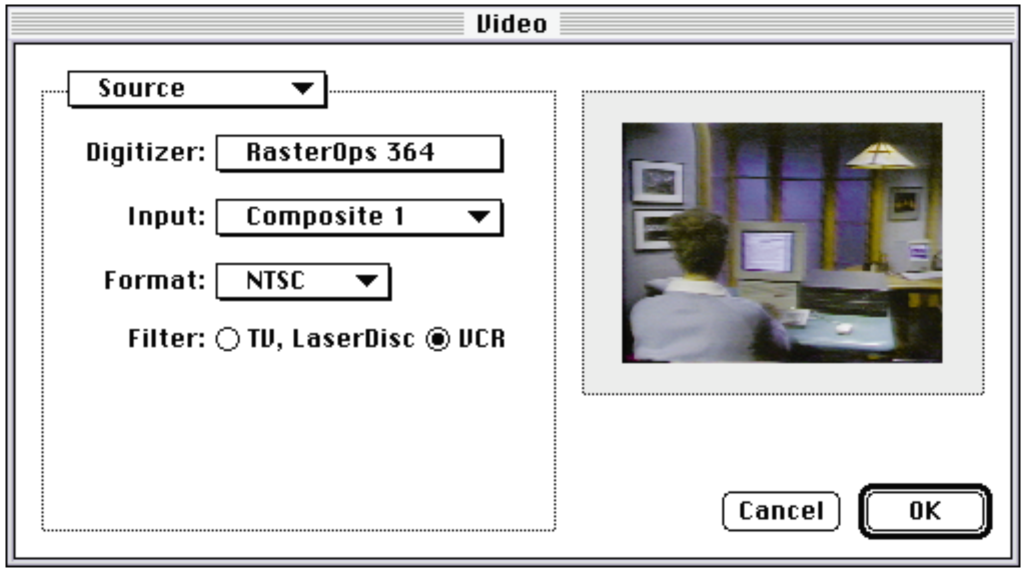
Typically, compressed-source devices are able to act as hardware decompressors and have a corresponding QuickTime image decompressor component that clients can use to play back the compressed images. However, if your hardware produces compressed data that can't be read by any of the standard QuickTime image decompressor components, you need to provide an appropriate software-only decompressor component. This way, users who don't have your hardware will still be able to play movies produced with your compressor.

A STANDARD USER INTERFACE FOR CONFIGURING VIDEO DIGITIZERS

QuickTime 1.5 introduces changes to the sequence grabber component, which many developers rely on to reduce the complexity of dealing with video and sound digitizers. One of the more significant additions to the sequence grabber component is standard dialog boxes for configuring video and sound digitizers. Both Apple and third parties can extend the controls presented in these dialog boxes through sequence grabber *panel components*. Although this article won't delve into the details of writing a panel component, you should be aware of them and how they relate to video digitizers.

Figure 6 shows the Source panel, one of the three panel components built into QuickTime 1.5. The other two panel components are the Image panel and the Compression panel. You navigate to the different panels through the pop-up menu at the top of the panel area.

If your video digitizer has capabilities that aren't addressed by the standard panels in QuickTime 1.5, and it's important to give users access to these features, we strongly



Digitizer: RasterOps 364

Input: Composite 1

Format: NTSC

Filter: ○ TV, LaserDisc ● VCR

Figure 6
The Source Panel

urge you to write a panel component. Any applications that use the sequence grabber dialog boxes can then pick up the functionality in your panel component. There's less work for everybody this way — video digitizer manufacturers don't have to write applications that show off their card-specific features, and application developers don't have to write card-specific code.

You'll find a description of how to write your own sequence grabber panel components in Chapter 6 of *Inside Macintosh: QuickTime Components*. As a bonus, we've included the source code for an example panel component on this issue's CD.

THE IDEAL VIDEO DIGITIZER

To bring a video digitizer board to market, developers must make numerous compromises between design, features, and cost. It's not unusual for a few hasty design decisions to lead to a less-than-wonderful QuickTime digitizer product.

For example, if the SuperOps company creates a video digitizing card that can do video play through at only 24 bpp in a display size of 640 by 480 pixels, and that can capture at only 8 bpp in a display size of 240 by 180 pixels, the card probably isn't going to be a prime candidate for the next QuickTime product of the year. Why? Because the video digitizer API is designed to encourage the creation of devices that have more or less symmetrical operating characteristics in both the play-through and capture modes. Obviously, the card just described doesn't meet this goal — it behaves quite differently depending on whether it's performing play through or capture. In addition, the choice of 240 by 180 pixels as the board's capture size is unwise. In general, an ActiveSrcRect size that's evenly divisible by 2 produces the best quality without the support of sophisticated filtering (see "What's So Magic About Magic Sizes?").

WHAT'S SO MAGIC ABOUT MAGIC SIZES?

When the vertical dimension of the ActiveSrcRect can be evenly divided by 2, the resulting sizes are referred to as *magic sizes.* Actually, there's nothing magic about them. They just happen to be easily produced by some simple tricks in hardware and software. It takes two fields to produce a complete frame of full-size NTSC video (for example, 640 by 480 pixels), with each field holding half the lines of the video frame. The most common resizing trick is to drop a field from each video frame to produce the requisite number of lines for a half-size image (for example, 320 by 240 pixels). This makes it very simple to produce good-quality half-size video without performing more sophisticated image filtering in software. Because software isn't burdened with the additional task of performing a more complex filtering of the image, the effective capture rate of the digitizer is increased at this "magic" size.

So how does a developer decide what sorts of features to include in a digitizer board? Well, we've got some pretty solid ideas about what characteristics make for a really great digitizer. To get a sense of the kind of features we believe an ideal QuickTime video digitizer should have, take a look at Table 1. Keep in mind that each of the features of this ideal digitizer has a very real associated cost, which gets reflected in the retail price. Our ideal digitizer would probably be priced out of the reach of most users today. But then again, two years ago we wouldn't have believed we could afford personal computers with 32 MB of RAM, so who knows?

Table 1
Characteristics of an Ideal Video Digitizer

Feature	**Benefit**
Hardware DMA support	The ability to display video on any screen (or off-screen) and not be captive to a local frame buffer
Enhanced resizing algorithms (anti-aliasing, improved line filtering, and so on)	Improved image quality
Arbitrary resizing in hardware	Fast resizing to any size
16-bit pixel support	Reduced data rate with minimal loss of color fidelity
Enhanced color control	Finer control over compression efficiency and improved image quality
Hardware compression and decompression	Improved live-capture frame rates, movie playback rates, and bigger movie sizes
Arbitrary hardware zoom (stretch, shrink)	Better cropping
Multiple buffering	Better performance
Signal lock sensing and standard detection	Improved user experience
Symmetrical play-through and record characteristics	Improved user feedback
Hardware special effects (flips, warps, skews)	Fun stuff to create that jaw-dropping impact on users
Key color	Fast masking, blue screen effects
Hardware clip mask	Video window clipping in graphics environments
Simultaneous compression and decompression of multiple video channels	Video conferencing and video phone applications

GO GRAB A MOVIE

Wow, we've covered a lot of ground here! Start thinking about integrating video data types into your applications, and experiment with the sample code to get started. The main message we want to pass on to application writers is that you should use the sequence grabber as your entry point into the world of video. Taking advantage of the enormous capability of the sequence grabber leaves you more time to spend on creating features that differentiate your product. As for video digitizer developers, we hope this article has given you more insight into the grab process and gotten you ready to start implementing some of the cool features listed in Table 1. We're counting on seeing many of these capabilities in the next generation of digitizer boards. Finally, to the video neophyte, all we can say is that there aren't too many things that are more entertaining than going out and grabbing a movie. So go do it!

REFERENCES

- *Inside Macintosh: QuickTime* and *Inside Macintosh: QuickTime Components*. These are included in the QuickTime Developer's Kit v. 1.5.
- "Inside QuickTime and Component-Based Managers" by Bill Guschwan, *develop* Issue 13.
- "Techniques for Writing and Debugging Components" by Gary Woodcock and Casey King, *develop* Issue 12.
- "Time Bases: The Heartbeat of QuickTime" by Guillermo A. Ortiz, *develop* Issue 12.
- *Electronic Cinematography: Achieving Photographic Control over the Video Image* by Harry Mathias and Richard Patterson (Wadsworth, 1985).
- *Film Art: An Introduction*, 3rd ed., by David Bordwell and Kristin Thompson (McGraw-Hill, 1990).
- *Graphics Gems* Volume I edited by Andrew S. Glassner (Academic Press, 1990), Chapter 3, "Useful 1-to-1 Pixel Transforms."
- *Raster Graphics Handbook*, 2nd ed., Conrac Corporation (Van Nostrand Reinhold, 1985).
- *Television Production*, 3rd ed., by Alan Wurtzel and Stephen R. Acker (McGraw-Hill, 1989).

THANKS TO OUR TECHNICAL REVIEWERS
Bill Guschwan, Peter Hoddie, Guillermo Ortiz, John Wang•

SOMEWHERE IN QUICKTIME

DERIVED MEDIA HANDLERS

JOHN WANG

In this column, I'll be telling you about derived media handlers in QuickTime 1.5 — but first, some background.

QuickTime movies contain tracks that refer to media. In QuickTime 1.0, two media types are supported: video and sound. A movie might therefore have one track that refers to video media and one that refers to sound media. Each of these supported media has a media handler, which is code that's responsible for interpreting the media's data. Obviously, displaying video images requires different code than playing sound. The media handler code is in the form of a component of type 'mhlr'. The video media handler has the subtype 'vide', and the sound media handler has the subtype 'soun'.

QuickTime uses the concept of a media to separate media interpretation from the Movie Toolbox and to place the responsibility into individual media handlers. This has the added advantage that media handlers can be created to interpret new media types. However, it wasn't possible to easily create a media handler in QuickTime 1.0.

DERIVED HANDLERS TO THE RESCUE

Derived media handler support was introduced in QuickTime 1.5 to allow developers to define new custom media types. As an example of the capabilities of the derived media handler, QuickTime 1.5 has a new 'text' media type that's implemented using a derived media handler. Derived media handler components can easily be created because they can use the services of a common base media handler supplied by Apple; hence the name *derived media handler*. The base media handler manages most of the duties that must be performed by all media handlers and reduces the intricacies of writing a standalone media handler.

This column will discuss sample code (provided on this issue's CD) that implements a complete QuickDraw derived media handler. This media handler will interpret QuickDraw pictures stored in the media's data. Each media sample in the data is a QuickDraw picture. For example, you could have a movie of a bouncing ball, but instead of having compressed pixel images of the balls bouncing, as in a video media, you would have a series of pictures of a ball drawn with PaintOval as it moves along its path. The CD also contains a sample that creates interesting movies using our new QuickDraw media type.

CREATING THE COMPONENT SHELL

The first step in creating our sample derived media handler is to create a component shell to which we can add media handler–specific calls in a later step. The MyComponent.c file contains the following routines: main (the dispatch routine for the component), MyOpen, MyClose, MyCanDo, MyVersion, and MyRegister.

MyOpen, the initialization routine for our component, opens the base media handler and sends a *target request* to it. A target request is a Component Manager service that allows a new component instance to establish itself as a *target* component instance for another (*delegate*) instance. In our case, the target is our QuickDraw derived media handler and the delegate is the base media handler. The delegate will be called by the target whenever the target wants to delegate calls to it. The delegate should call the target whenever the delegate would normally call itself (for example, when it uses its own services). This effectively makes our derived media handler sit on top of the base media handler by

JOHN WANG (AppleLink WANG.JY) of Apple's Printing, Imaging, and Graphics group was once a math and science nerd whose writing skills were as bad as the BASIC programming language compared to C. His hard work (hah) and prep school training finally pulled him through. Yet he still can't believe he's writing to an audience greater than just himself. (The editors can't either.) He's even got a double feature in this issue (see also "Print Hints"). Will wonders never cease.•

For more information about derived media handlers, see the Derived Media Handler Components chapter of *Inside Macintosh: QuickTime Components* (which is included in the QuickTime Developer's Kit v. 1.5).•

handling all requests that it can handle and delegating requests that it can't handle to the base media handler.

By calling the ComponentSetTarget routine after opening an instance of the base media handler component, we inform the base media handler that our component is derived from it. For example, the following code from our derived media handler component's open routine, MyOpen, will open the base media handler and target it:

```
myComp = OpenDefaultComponent(MediaHandlerType,
      BaseMediaType);
ComponentSetTarget(myComp, self);
(**storage).delegate = myComp;
```

The above description of targeting a component is similar to another Component Manager service, called *capturing*. Capturing is a service whereby the target component completely and permanently overrides the delegate component by hiding it from further use. This is feasible, for example, when updating a component or fixing bugs in it; the target can implement only the new features while delegating the original functionality to its delegate. Since the target wouldn't want the outdated delegate component to be visible any longer, it would capture the delegate using the Component Manager's CaptureComponent routine. You shouldn't call CaptureComponent on the base media handler, because that would hide it and prevent other derived media handlers from using it. Conceptually, you're not replacing the base media handler; you're just using its services. Therefore, targeting it is sufficient.

When the media handler is no longer used by the Movie Toolbox, the QuickDraw derived media handler's MyClose routine will be called. To close the connection to the base media handler, MyClose must call CloseComponent to close the base media handler component instance:

```
CloseComponent((**storage).delegate);
```

To prevent our derived media handler from registering if the base media handler isn't available, the MyRegister routine returns true (to not register) if the initialization done by MyOpen fails or false (to register) if it succeeds.

DEFINING THE MEDIA DATA FORMAT

The next step in creating a derived media handler is to define a media type identifier, a sample description record that's stored along with the media samples, and the format of the media data.

For our QuickDraw media, we'll use 'Qdrw' as the media type. (As with resource types, Apple reserves all-lowercase types, so we use a media type that contains one uppercase character.) Every movie that contains a track created by using NewTrackMedia with mediaType 'Qdrw' will automatically refer to our custom media handler. Our media handler will have the component type 'mhlr' and subtype 'Qdrw'.

All description records must contain size, type, resvd1, resvd2, and dataRefIndex fields as a minimum. You should always fill in the size and type fields, but you can set the other fields to 0. It's also recommended that a field for the media data version be included in the sample description record so that it's always possible to identify the version of the media data.

```
#define  QDrawMediaType 'Qdrw'
#define  QDMediaVersion 0x100
typedef struct GraphicsDescription {
   long  size;
   long  type;       // QDrawMediaType
   long  resvd1;
   short resvd2;
   short dataRefIndex;
   short version;    // QDMediaVersion
} QDrawDescription, *QDrawDescriptionPtr,
   **QDrawDescriptionHandle;
```

Since every media data sample has an associated description record, it's possible to find out the version, or any other data defined in the record, for the particular sample. To prevent wasting space with duplicate description records, QuickTime associates a sample description index with each sample; thus, many

For more information on the Component Manager, see "Techniques for Writing and Debugging Components" in *develop* Issue 12 and the Component Manager documentation in the QuickTime Developer's Kit v. 1.5. (The Component Manager will also be described in *Inside Macintosh: More Macintosh Toolbox*.)•

media samples can refer to the same description record through its index.

As mentioned earlier, the samples stored in our media data are simply QuickDraw pictures. So the data handle passed to AddMediaSample, which is the Movie Toolbox call to add data into a media, will simply be a PicHandle. This allows us to easily create sample data with OpenPicture and ClosePicture and dispose of it with KillPicture.

CREATING A MOVIE THAT USES THE DERIVED MEDIA HANDLER

When we start writing the implementation-specific component routines for the media handler, we'd like to be able to test them with a QuickTime movie that uses that media handler. So, the next step is to write an application that creates a movie using the QuickDraw media handler. But there's a sort of Catch-22: We won't be able to run the application to create the movie until after we begin implementing some of the QuickDraw media handler component routines, because some of the Movie Toolbox calls that are used to create the movie will use the QuickDraw media handler. For example, NewTrackMedia will cause QuickTime to open an instance of our component to prepare for editing and playback of the new media.

The code below shows how to create a movie using our newly defined QuickDraw media format (the complete code is on this issue's CD). AddGraphics, defined later, is a wrapper procedure for AddMediaSample that any application can call to easily add media samples.

```
// Create track and media.
myTrack = NewMovieTrack(myMovie,
            (long) kFrameWidth << 16,
            (long) kFrameHeight << 16, 0);
myMedia = NewTrackMedia(myTrack, QDrawMediaType,
         600, nil, (OSType) nil);

// Add samples to media.
BeginMediaEdits(myMedia);
myQDDesc = (QDrawDescriptionHandle)
      NewHandleClear(sizeof(QDrawDescription));
(**myQDDesc).size = sizeof(QDrawDescription);
(**myQDDesc).type = QDrawMediaType;
(**myQDDesc).version = QDMediaVersion;
myPict = OpenPicture(&drawRect);
PaintOval(&drawRect);
ClosePicture();
AddGraphics(myMedia, myPict, myQDDesc, 600, 0,
         nil);
DrawPicture(myPict, &drawRect);
KillPicture(myPict);
EndMediaEdits(myMedia);

// Place media into movie.
InsertMediaIntoTrack(myTrack, 0, 0,
         GetMediaDuration(myMedia), kFix1);
```

The main difference between code that generates a movie using normal QuickTime video media and code that uses our QuickDraw media is in the NewTrackMedia and AddMediaSample calls. For NewTrackMedia, we pass 'Qdrw', as defined earlier, for the mediaType parameter.

The wrapper procedure AddGraphics is defined as follows:

```
pascal OSErr AddGraphics(Media graphicsMedia,
                  PicHandle myPic,
                  QDrawDescriptionHandle QDDesc,
                  TimeValue duration,
                  short mySync;
                  TimeValue *sampleTime)
{
   return (AddMediaSample(graphicsMedia,
      (Handle) myPic, 0L, GetHandleSize(myPic),
      duration, (SampleDescriptionHandle)
      QDDesc, 1L, mySync, sampleTime));
}
```

ADDING THE MEDIA HANDLER ROUTINES

The last step is, of course, to complete our derived media handler component.

The file named MyMediaComponentRoutines.c contains the routines that our QuickDraw media

For more information on the picture format, see the "Color Picture Format" section of the Color QuickDraw chapter of *Inside Macintosh* Volume V.•

handler implements rather than delegating to the base media handler. It wouldn't make much sense to delegate MediaInitialize and MediaIdle, since these routines are crucial to our code: MediaInitialize initializes our media handler and MediaIdle is the routine that gets called for drawing. On the other hand, a call such as MediaGSetVolume wouldn't be very useful to our very quiet graphics media, so MediaGSetVolume would be delegated to the base media handler.

All media handler routines, as defined in the Derived Media Handler Components chapter, must be delegated to the base media handler if not implemented. We've chosen to implement the following routines because our media handler does spatial processing (in other words, it draws).

- MediaInitialize: Prepares access to media by saving necessary information passed to it in the GetMovieCompleteParams record.
- MediaIdle: The Movie Toolbox provides processing time to the derived media handler through this routine. The QuickDraw media handler draws during this call.
- MediaSetActive: The Movie Toolbox calls this routine if the media is enabled or disabled.
- MediaSetRate: Called if the rate changes. This is necessary to determine when the movie rate is reversed.
- MediaSetMediaTimeScale: Our media handler cares about the media time scale since we store times in the media's time coordinate system.
- MediaTrackEdited: Called if the track has been edited. If so, we'll want to redraw.
- MediaSetGWorld: Called if the destination GWorld changes. If so, we'll want to know the new GWorld for drawing.
- MediaSetDimensions: Called if the spatial dimensions change.
- MediaSetMatrix: Called if the track matrix or movie matrix changes due to movie resizing.
- MediaGetTrackOpaque: Called to determine whether the track is opaque. We want to return true so that correct compositing occurs. Our media may be semitransparent.
- MediaSampleDescriptionChanged: Called if the sample description record changes. If we ever store information beyond the media data version, we'll probably want to know if the user changes the sample description contents. If the description record changes due to different media samples referring to different description records, this routine isn't called. Instead, the media handler must check the sample description index returned by GetMediaSample.

The routines we didn't implement are:

- MediaGGetStatus: Our simplified QuickDraw media handler doesn't need any error processing.
- MediaPutMediaInfo, MediaGetMediaInfo: Since we don't store any proprietary information along with the media data, we don't need to implement this.
- MediaSetMovieTimeScale: Our media handler doesn't care about the movie time scale since we don't store any times in the movie's time coordinate system.
- MediaSetClip: Our media handler doesn't support clipping.
- MediaSetGraphicsMode, MediaGetGraphicsMode: Our media handler doesn't support graphics modes.
- MediaGetNextBoundsChange: Our bounds doesn't change dynamically. (The text media handler is an example of a media that has dynamically changing bounds.)
- MediaGetSrcRgn: Our media doesn't have an irregular display region.
- MediaGSetVolume, MediaSetSoundBalance, MediaGetSoundBalance: We don't play sound.
- MediaPreroll: We let the base media handler do our prerolling for us. The data handler is smart about caching, so we really don't need to worry about this.

Since it's difficult for us to cover every possible condition in which the media handler will get called, a clever approach is needed to aid in the development of the derived media handler. The solution lies in DebugStr: By strategically placing DebugStr calls throughout the media handler, we can see which routines are being called. Knowing which events trigger calls to our media handler will allow us to decide which calls the handler should support and which ones we can delegate. For example, it was through this process that I found that MediaSetMatrix was a call I needed to implement because resizing a movie window causes MediaSetMatrix to be called.

This approach makes it possible to create a media handler starting off with just a MediaInitialize routine and building from there. The selectors for a media handler component are in the range of 0x500 to 0x5FF. Therefore, any calls that are delegated with selectors in this range should be examined to determine whether the actions that cause the routine to be called are significant to the media handler implementation. If so, the routine should not be delegated. The QuickTime documentation, this column, the sample code, and common sense should give you a good idea of which media handler–specific routines a derived media handler must implement.

THE GUTS OF THE MATTER

As you can see in MyMediaComponentRoutines.c, most of the guts are in the routines MediaInitialize and MediaIdle. MediaInitialize is called by the Movie Toolbox when a movie using the media is opened. The MediaInitialize routine should grab information it needs that's passed to it by QuickTime in the GetMovieCompleteParams record and store it in a private data structure.

```
typedef struct {
   short          version;
   Movie          theMovie;
   Track          theTrack;
   Media          theMedia;
   TimeScale      movieScale;
   TimeScale      mediaScale;
   TimeValue      movieDuration;
   TimeValue      trackDuration;
   TimeValue      mediaDuration;
   Fixed          effectiveRate;
   TimeBase       timeBase;
   short          volume;
   Fixed          width;
   Fixed          height;
   MatrixRecord   trackMovieMatrix;
   CGrafPtr       moviePort;
   GDHandle       movieGD;
   PixMapHandle   trackMatte;
} GetMovieCompleteParams;

typedef struct {
   // Component stuff
   ComponentInstance delegate;
   ComponentInstance self;

   // Characteristics
   Movie          myMovie;
   Track          myTrack;
   Media          myMedia;
   Fixed          mediaRate;
   Rect           graphicsBox;
   MatrixRecord   myMatrix;
   CGrafPtr       port;
   GDHandle       device;
   long           sampleDescIndex;

   // Media globals
   long           somethingChanged;
   Boolean        enabled;
   Fixed          newMediaRate;
   TimeValue      lastMediaTime;
} PrivateGlobals;
```

The above private data structure for the QuickDraw derived media handler shows the fields that the handler is interested in. For example, we would obviously need to know the trackMovieMatrix, but a sound media handler would not. The information in the GetMovieCompleteParams record is valid at the time of the MediaInitialize call and is updated through other derived media handler routines such as

MediaSetGWorld. It's important to implement such derived media handler routines to update information used by the media handler.

MediaInitialize also needs to inform the base media handler of its capabilities by calling the routine MediaSetHandlerCapabilities. Our media handler uses this routine to tell the base media handler that we perform spatial processing and that we also can work with transfer modes.

MediaIdle does the bulk of the processing in a media handler. Our MediaIdle routine uses GetMediaSample to get the media sample and then calls DrawPicture to display the sample. It also uses a scheme of calling GetMediaNextInterestingTime to implement sync frames; this allows greater performance when playing movies backward, because the media handler won't have to begin drawing from the beginning of the media. The concept of sync frames is important in QuickTime movies because it allows temporal compression so that not all frames need to contain complete state information. Keeping a small number of frames between key frames makes it possible to preserve performance when playing movies backward, since rendering of frames must still occur in the forward direction.

The effect of not having sync frames is evident if you create a movie without any (see the example on the CD). Such a movie will look to the media handler like a movie in which every frame is a key frame. As you can see with these movies, backward playback of movies gives different results than forward playback since each sample is treated as a sync sample even though it shouldn't be. This is not recommended because it's conceptually and visually confusing to users.

In addition, to prevent the redrawing of previously drawn frames, our media handler keeps track of the last media time that the image has been updated so that it can continue from there if no other changes to the environment prevent it. This works only when the movie is playing in the forward direction. When a movie is played backward, each frame must be completely recreated starting from the last key frame.

OUT OF TIME

Using the QuickDraw derived media handler as a framework, you can create your own media type. Interactivity tracks, custom sound format tracks (such as MIDI), and even hardware control tracks are all possible. With some creativity and work, you can expand the capabilities of QuickTime beyond imaginable limits.

Thanks to Ken Doyle, Bill Guschwan, Peter Hoddie, and Guillermo Ortiz for reviewing this column.•

MAKING BETTER QUICKTIME MOVIES

QuickTime 1.5 makes it easier than ever to make CD-playable movies. These tips on capturing, compressing, and playing back movies will help you use the new Apple Compact Video compressor to its best advantage, creating movies that will play well off a standard CD-ROM drive on a Macintosh LC computer.

KIP OLSON

QuickTime introduced the world of digital video to the Macintosh and enabled a whole new category of multimedia content: movies. With QuickTime, it's simple to play back movies in any application and to exchange movies between applications using the standard cut/copy/paste mechanism. But no one ever said it was going to be easy to create them!

What makes movie creation tricky is the tradeoffs required to get QuickTime movies to play off a CD-ROM drive, the most effective distribution medium for digital video. Uncompressed, full-screen, full-motion video requires a data transfer rate of about 27,000 kilobytes/second, yet a typical CD-ROM drive has a data transfer rate of only 100 kilobytes/second. QuickTime solves this problem by using video compression, which requires you to make tradeoffs between frame size, frame rate, image quality, and sound quality when making a movie.

The tips in this article will help you make the right tradeoffs to produce high-quality movies that will play off CD-ROM. You'll also find tips on capturing digital video, using the MovieShop utility on this issue's CD to produce compressed movies, and playing back what you've created. I assume you're already familiar with the basics of movie making with QuickTime.

TIPS ON TRADEOFFS

Making the right tradeoffs is the key to producing better QuickTime movies. Depending on your target platform, to get smooth playback you may need to limit the frame size and rate, minimize the differences between frames, and trade off audio for video quality.

KIP OLSON lives by the motto "I never met an avalanche I didn't like" and spends his winters skiing the extreme in places like Chamonix, Jackson Hole, and Hoboken. When not cornice surfing, he's been known to cast a #12 Adams to a rising brookie and score a perfect 18 at putt-putt golf. He works for Apple in his spare time.•

PLATFORM

Before you get started, you need to decide which Macintosh platform your movie will play on. Obviously, a Macintosh Quadra 950 with a double-speed CD-ROM drive can play much larger, higher-quality movies than a Macintosh LC with a standard CD drive. Table 1 shows some common platforms and their capabilities. (See the following sections for more on frame size and rate.)

Table 1
Common QuickTime Platforms

Macintosh	Bits/Pixel	Built-in CD Drive	Maximum Movie Frame Size, Rate	Market
Quadra 950	1, 4, 8, 16, 24	Optional	320 x 240, 24 fps	Power user, professional
Performa 600	1, 4, 8, 16	Optional	320 x 240, 15 fps	Consumer, home
LC II	1, 4, 8, 16	No	240 x 180, 12 fps	Education
Macintosh II	Card-dependent	No	160 x 120, 15 fps	Loyal customers

For the purposes of this article, our target platform is the Macintosh LC II with an AppleCD SC CD-ROM drive (transfer rate of 100 kilobytes/second). Movies created for this platform should play back well on virtually every color Macintosh, covering as much of the installed base as possible. However, keep in mind that machines that use NuBus video, such as the Macintosh II, won't have the playback performance of the LC II. You should always test your movies on the platforms they'll play on.

FRAME SIZE

The frame size determines how large the movie will be on the screen. The larger the frame size, the greater the number of pixels that have to be updated every frame. This can be a problem for less powerful machines, so you often need to limit the frame size to get smooth playback.

Frame sizes are typically specified by horizontal and vertical pixel measurements. Some common frame sizes for digital video are shown in Table 2.

Table 2
Common Frame Sizes for Digital Video

Frame Size	Description	Pixels/Frame	Capability Required
640 x 480	Full-screen	307,200	Hardware acceleration
320 x 240	Quarter-screen	76,800	Fast CPU like a Macintosh Quadra
240 x 180	Eighth-screen	43,200	Apple Compact Video compressor
160 x 120	Sixteenth-screen	19,200	Apple Video compressor

For more information on QuickTime 1.5, your best source is the QuickTime Developer's Kit v. 1.5, available from APDA. This kit includes the complete printed documentation for QuickTime 1.5 as well as the *QuickTime 1.5 Developer CD*, which is full of example code, movies, and utilities. The new *Inside Macintosh: QuickTime* and *Inside Macintosh: QuickTime Components* together provide a complete description of QuickTime 1.5.•

Note that full-screen movies are practical only with hardware acceleration, and for quarter-screen movies you need a fast CPU like a Macintosh Quadra. With our LC II platform and the Apple Compact Video compressor made available by QuickTime 1.5, we can create eighth-screen movies, which have more than twice the screen area of the "postage-stamp movies" possible with QuickTime 1.0's Apple Video compressor.

For the Apple Compact Video compressor to function optimally, the frame size should be a multiple of 4 in each dimension. This is because the compression algorithm uses a 4-pixel by 4-pixel cell.

FRAME RATE

The *frame rate* is the number of frames displayed in each second of the movie, typically described in frames per second (fps). The frame rate to use for a movie depends on the frame rate of the source material, whether film or videotape. For the smoothest results, you should use a frame rate of which the source material frame rate is a multiple, but this may only be possible if you have hardware acceleration or a fast CPU. Still, an acceptable compromise is available if your platform is limited.

The frame rate of source material in the NTSC video format is approximately 30 fps. Much source material is shot using film at 24 fps and then transferred onto videotape. Frame rates to use for movies based on these types of source material are shown in Table 3. Other video standards such as PAL and SECAM have different frame rates; if your movie is based on one of these types of source material, you'll have to compensate accordingly. For our target platform, we can use 12 fps with good results.

There are a couple of minor quirks having to do with frame rate that you should be aware of when you make a movie. First, you'll note that I said the NTSC frame rate

Table 3
Common Frame Rates for Digital Video

If Your Source Material Is NTSC Video:

Frame Rate	Description	Capability Required
30 fps	Full-motion	Hardware acceleration
15 fps	Half-motion	Fast CPU like a Macintosh Quadra
12 fps	Half-film rate	Apple Compact Video compressor
10 fps	Third-motion	Apple Video compressor

If Your Source Material Is Film:

Frame Rate	Description	Capability Required
24 fps	Full-motion	Hardware acceleration
12 fps	Half-motion	Apple Compact Video compressor
10 fps	Third-video rate	Apple Video compressor

is *approximately* 30 fps. For reasons lost in the dawn of television, the NTSC frame rate is actually 29.97 fps. If you assume the frame rate is 30 fps, long movies can lose synchronization between sound and video over time, since there are fewer video frames than expected. For example, if you digitized 100 seconds of video, you would expect to get 3000 frames, but you would really only get 2997 frames in that period of time. The GrabGuy utility and the HyperCard® Movie Making Stack (found on the *QuickTime 1.5 Developer CD* and on this issue's CD) automatically take care of this problem, but if you find sound sync drifting over time on long movies, you may need to duplicate a video frame every 1000 frames to get things back in sync.

The second item to note involves transferring 24-fps film to 30-fps video. On videotape, each frame is composed of two fields, one containing the odd scan lines and the other containing the even scan lines. These fields are interlaced to produce the frame. When film is transferred to video, six extra frames are "made up" every second. Typically, once every four frames, two adjacent film frames are put into the two fields of a single video frame to form a fifth frame. Figure 1 shows how this works. These made-up frames have a blurred look when digitized.

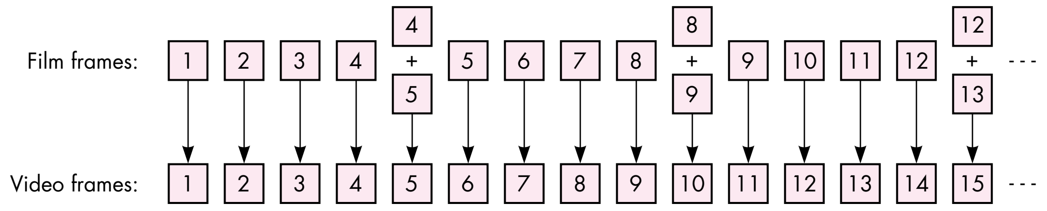


Figure 1
Making Up Extra Frames When Film Is Transferred to Video

You can use a couple of different methods to digitize only the original film frames and skip the blurry made-up frames. If you have a capture system that can grab individual video fields, you can set it to capture at 12 fps, and it will skip the duplicate fields and give the original 12 film frames each second. Or simply capture at 30 fps and throw away every fifth frame, yielding the original 24 film frames.

FRAME DIFFERENCING

Frame differencing is the technique used by QuickTime of storing and updating only the pixels that differ from the previous frame, so that much less data has to be stored and displayed. For example, in Figure 2 the frame on the right contains only the information needed to update the areas of the screen that differ from the frame on the left. As a consequence, less data has to be stored on disc for the second frame and it takes less time to draw. This in turn allows larger frame sizes and frame rates, giving better-quality movies.

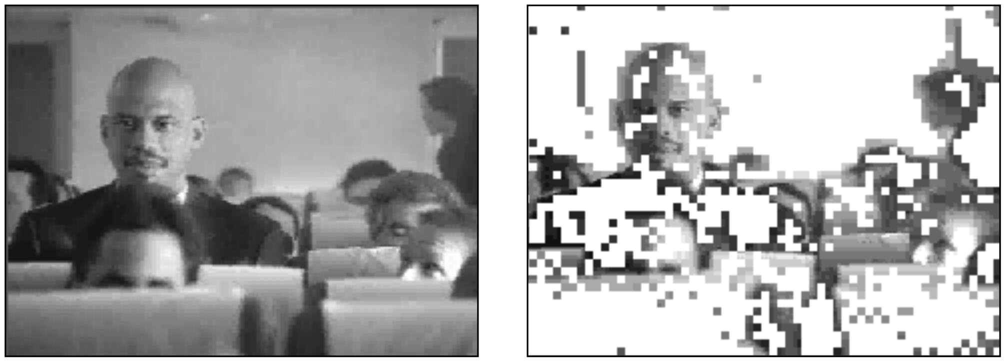


Figure 2
Example of Frame Differencing

When you're using the Apple Compact Video compressor, it's a good idea to create movies in which not much changes from one frame to the next, since frame differencing is one way the compressor achieves lower data transfer rates. Here are some things to keep in mind to get the most benefit from frame differencing:

- When possible, use source material with constant backgrounds and solid colors — especially all-black and all-white areas — to reduce the difference between frames.
- When possible, use videos of "talking heads." These are great candidates for frame differencing, since typically only the lips and head move.
- Avoid videos with lots of panning and zooming or with complex backgrounds. These effects increase the difference between frames and thus decrease the possibility of compression gains.
- Avoid source material with a lot of video noise, as this increases the difference between frames.

Frames can't be differenced indefinitely, however. At regular intervals a *key frame* — a frame that refreshes the entire movie area, not just the pixels that differ from the previous frame — is inserted. You can adjust the interval to achieve the tradeoff between data rate and movie quality that you desire. Normally, you should have one key frame per second. The more you put in, the higher the data rate and the better your movie looks but the less compression gain you get from frame differencing.

SOUND SAMPLING RATE

The standard Macintosh sound rate is 22.254 kHz, so you'll get the highest-quality audio by sampling to this rate. The Sound Manager is also more efficient at this native rate, so the movie will play back better. However, because your data transfer rate off CD-ROM is limited, using higher audio rates will decrease the quality of the video. The same constraint applies to stereo and 16-bit sound, both of which are supported by QuickTime: these formats eat up more bandwidth, so the video quality may suffer.

If your source is mostly people talking and not music, you can record audio at the alternate sampling rate of 11.127 kHz. In many cases this will make your video look sharper and give acceptable sound quality.

With most audio sampling hardware, it's best to set the hardware at the sampling rate you desire, since the hardware will often do filtering to avoid aliasing artifacts. One exception is the MacRecorder digitizer (from MacroMedia), which doesn't filter at 11.127 kHz. If possible, you should record at 22.254 kHz on a MacRecorder and downsample the audio to 11.127 kHz when you compress the movie.

One other thing to keep in mind when using a MacRecorder to digitize audio at 22.254 kHz is that its sample rate tends to drift away from the standard Macintosh rate, so you should always resample it to 22.254 kHz or 11.127 kHz when you compress the movie.

TIPS ON CAPTURING

The *QuickTime 1.5 Developer CD* and this issue's CD contain two utilities for capturing video: GrabGuy and the HyperCard Movie Making Stack. GrabGuy is an application that does a multipass grab off a controllable VCR like the Sony µMatic, giving frame-accurate recording. HyperCard and the Movie Making Stack enable you to get frame-accurate grabs off controllable Pioneer laser discs. Most video cards also come with software that enables you to grab raw video directly to RAM or hard disk.

To get the highest possible quality when you're capturing the source material, you should do three things:

- Start with a clean source.
- Adjust black, white, brightness, and contrast levels.
- Grab at a larger size than you need and scale down when you compress.

I'll discuss these tips one at a time.

START WITH A CLEAN SOURCE

The less video noise, the better compression and the more benefit from frame differencing you'll get, so you should digitize from the cleanest, highest-quality video source possible. The most common video formats, in decreasing order of quality, are BetaCam, µMatic/S-VHS/Hi8, laser disc, and VHS.

For more on capturing, see the article "Video Digitizing Under QuickTime" in this issue.•

Beware of tapes that have been duplicated many times or played a lot — they can be very noisy. If your digitizing card supports S-Video inputs, use them if you can, as S-Video delivers better quality than composite video.

ADJUST LEVELS

Many digitizing cards support one or more settings with regard to black level, white level, brightness, and contrast. To enhance compression gains from frame differencing, you should adjust the black level of your card so that black areas in your source digitize as truly black pixels. A frame with truly black pixels differs much less from the original than the same frame with noisy black pixels. Thus, refreshing the screen with the noisy frame takes more data than refreshing the screen with the clean frame. Often what looks like black is quite noisy, so you should experiment with your video card. The same rule applies to white levels.

GRAB AT A LARGER SIZE THAN YOU NEED

Because many video cards do a poor job of scaling down frames when they grab, it's best to capture at a large size and let QuickDraw do a filtered scale when you compress. If you're using GrabGuy you don't need to worry, because it will grab 320 x 240 fields and use QuickDraw to scale them down. If you're grabbing from laser disc, grab frames at 640 x 480 and scale them down at compression time.

To save disc space, you should grab using JPEG compression set to the highest quality instead of grabbing raw frames. Most of the quality you might lose this way isn't used by the Apple Compact Video compressor anyway, so this won't reduce the quality of your final movie very much. A number of video cards now support hardware JPEG compression, which makes this even easier.

TIPS ON COMPRESSING

The MovieShop utility on this issue's CD is indispensable for compressing QuickTime movies. After an admonition to edit before compressing, I'll take you through the steps involved in using MovieShop to create CD-playable movies. My emphasis here is more on what to do than why you're doing it. If you're curious about the reasons you go through the steps you do, refer to the MovieShop documentation on the CD.

EDIT BEFORE COMPRESSING

To achieve the best possible playback performance with the smallest amount of memory, you should completely edit your movie in raw form before compressing with MovieShop. By the same token, it's not a good idea to compress the movie in pieces and then cut and paste the pieces together to form the final movie, as this will require extra input/output buffers and may cause QuickTime to run out of memory. If this happens, movie playback will slow down considerably.

Here's why: Movies are composed of tracks that typically contain video and sound data. When you capture a movie, the track data is often stored sequentially in the file, resulting in a file layout like the first one shown in Figure 3. To play this movie, QuickTime must allocate four large buffers — one for each track — and seek between tracks during playback. This can cause miserable playback performance off CD-ROM drives, which typically have very slow seek times (we're talking hundreds of milliseconds).

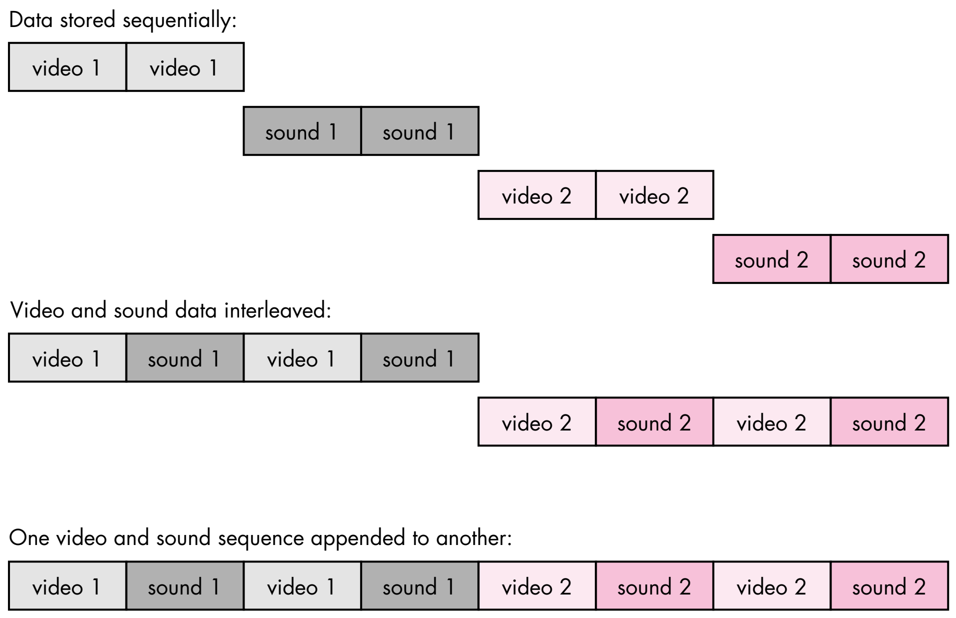


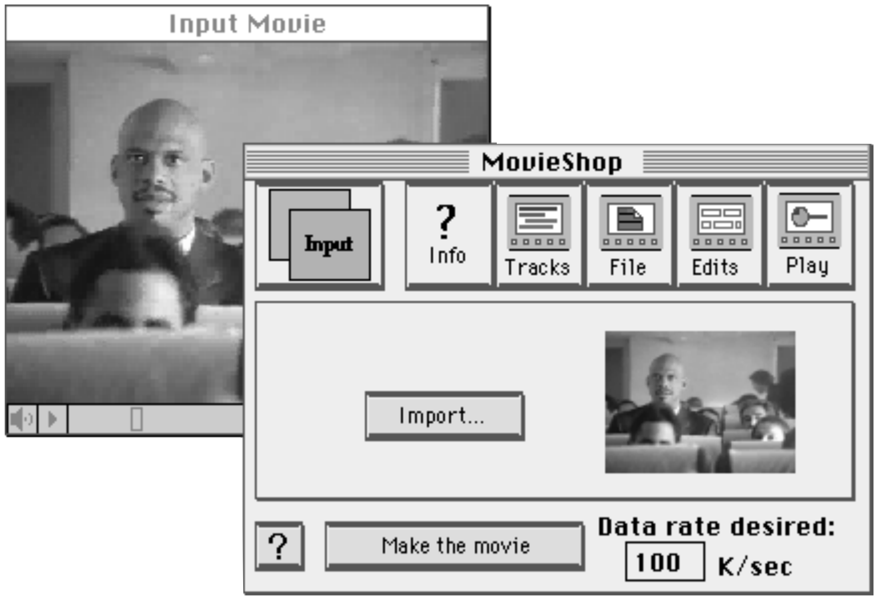
Figure 3
Ways of Storing Video and Sound Track Data in a Movie

To solve this problem, QuickTime enables you to interleave the video and sound tracks, resulting in a file layout like the second one shown in Figure 3. Because the sound and video track data are now close to each other in the file, seeking is minimized and only two buffers are needed for data transfer. This is often the file layout you get when you paste two movies together.

The most efficient layout is to append the second video/sound sequence onto the first, as illustrated by the third file layout in Figure 3, so that only one buffer is needed and playback is optimized. When you compress with MovieShop, it automatically merges all the video and sound tracks of a movie into a single video and sound track, thus giving you the most efficient layout.

SET THE DATA RATE

After importing a movie to compress, the first thing you do in MovieShop is to set the data rate. (See Figure 4.) To play off CD, a movie must be compressed to deliver a consistent data rate of 100 kilobytes/second or less, the effective data transfer rate of the first generation of CD-ROM drives. Some of the newer CD-ROM drives can now deliver twice this data rate, but you'll probably want to make your movie at 100 kilobytes/second anyway, for backward compatibility. So a data rate of 100 is usually best for CD playback, while a range from 90 to 105 kilobytes/second will usually produce good results.

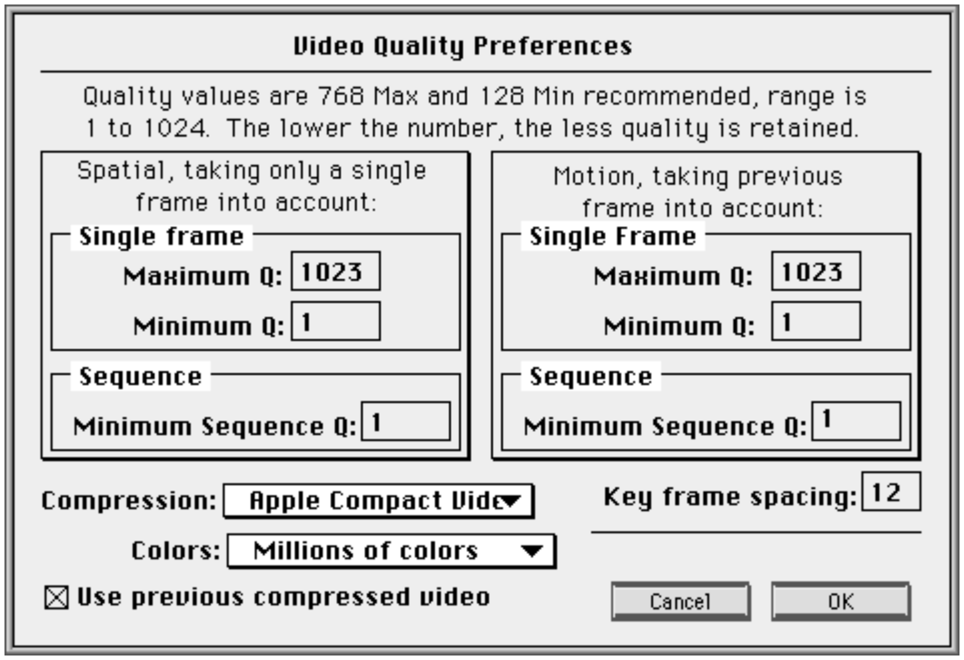


Data rate desired: 100 K/sec

Figure 4
Setting the Data Rate in MovieShop

SET THE VIDEO SETTINGS

Next you indicate your preferences relating to compression method, colors, and key frame spacing. To do so, choose Video from the Preferences menu. The dialog box shown in Figure 5 will be displayed. This is where you choose Apple Compact Video as your compression method. The Apple Compact Video compressor has been optimized for CD playback and has a built-in data rate constraining algorithm to give consistent playback from CD. Computationally, it's a highly asymmetric algorithm, taking about an hour to compress a minute of video. The results are worth it, however.


Video Quality Preferences

Quality values are 768 Max and 128 Min recommended, range is 1 to 1024. The lower the number, the less quality is retained.

Spatial, taking only a single frame into account:

Single frame
Maximum Q: 1023
Minimum Q: 1

Sequence
Minimum Sequence Q: 1

Motion, taking previous frame into account:

Single Frame
Maximum Q: 1023
Minimum Q: 1

Sequence
Minimum Sequence Q: 1

Compression: Apple Compact Vide
Colors: Millions of colors
☒ Use previous compressed video
Key frame spacing: 12

Figure 5
MovieShop's Video Quality Preferences Dialog

MovieShop tries to limit the data rate of a movie by adjusting the amount of frame differencing (motion quality) and lowering the compression quality (spatial quality). However, since the Apple Compact Video compressor determines motion and spatial quality internally, you should turn these settings off by entering the numbers shown in Figure 5. (For details about what these magic numbers mean, see the MovieShop documentation.) That way, Apple Compact Video will always make the right choices for the data rate you've chosen.

If your source material is in color, choose "Millions of colors"; if it's in black and white, choose "256 grays." "Use previous compressed video" should be checked for most video sources; however, if you're compressing raw animations or composite movies with a constant background, uncheck this setting to get more benefit from frame differencing. MovieShop will then use the last uncompressed frame instead of the last compressed frame as the basis for frame differencing.

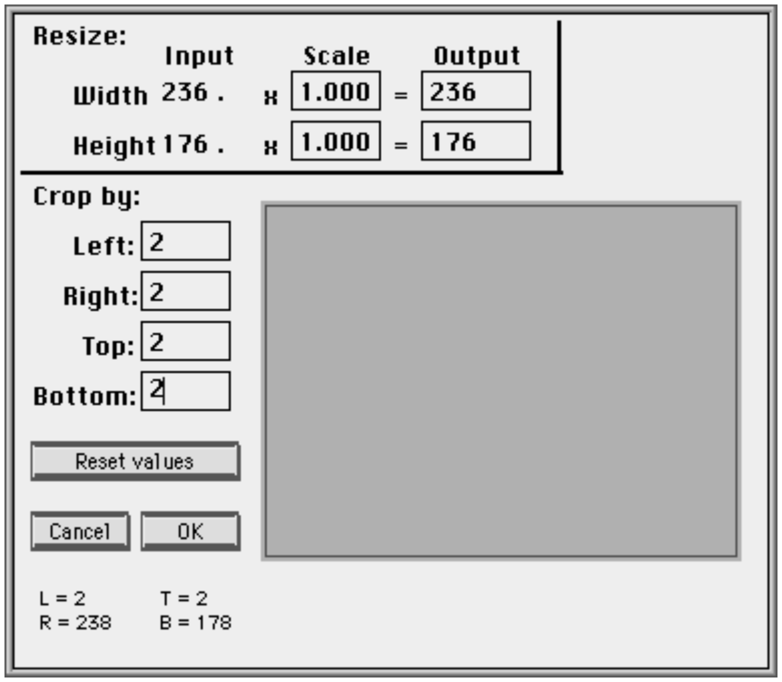
The key frame setting should be related to the frame rate of your video. As I mentioned earlier, you should normally have one key frame per second, although in some cases you might want to have fewer than one per second to lower the data rate (which will, however, decrease quality as well).

SET THE SOUND SETTINGS

To change sound settings, choose Sound from the Preferences menu. Selecting 22 kHz will ensure that your movie uses the standard Macintosh rate of 22.254 kHz. As discussed earlier, if you're not concerned about audio quality you should probably resample to 11 kHz as a tradeoff for sharper video. The Video to Sound setting lets you set how far ahead in seconds the audio is interleaved on the file from the video on the disc. This setting should normally be at 1.90, but if you find that CD playback is choppy or the audio portion breaks up, try lowering this number to 1.5. "Interleaved sound" should always be checked so that the audio and video are interleaved as explained earlier for smooth CD playback.

SCALE AND CROP THE MOVIE

Now you need to crop the movie, since there's often tape noise and jitter on the edges of the video frames. At the same time you can scale the movie to ensure that the frame size is a multiple of 4 pixels in each dimension. (Recall that due to its algorithm, the Apple Compact Video compressor functions optimally if this is so.) To change cropping settings and scale the movie, choose Cropping from the Preferences menu. In Figure 6, the movie is being cropped by 2 pixels on all sides to eliminate noisy edges and make the output dimension values multiples of 4.



Resize:

	Input		Scale		Output
Width	236 .	x	1.000	=	236
Height	176 .	x	1.000	=	176

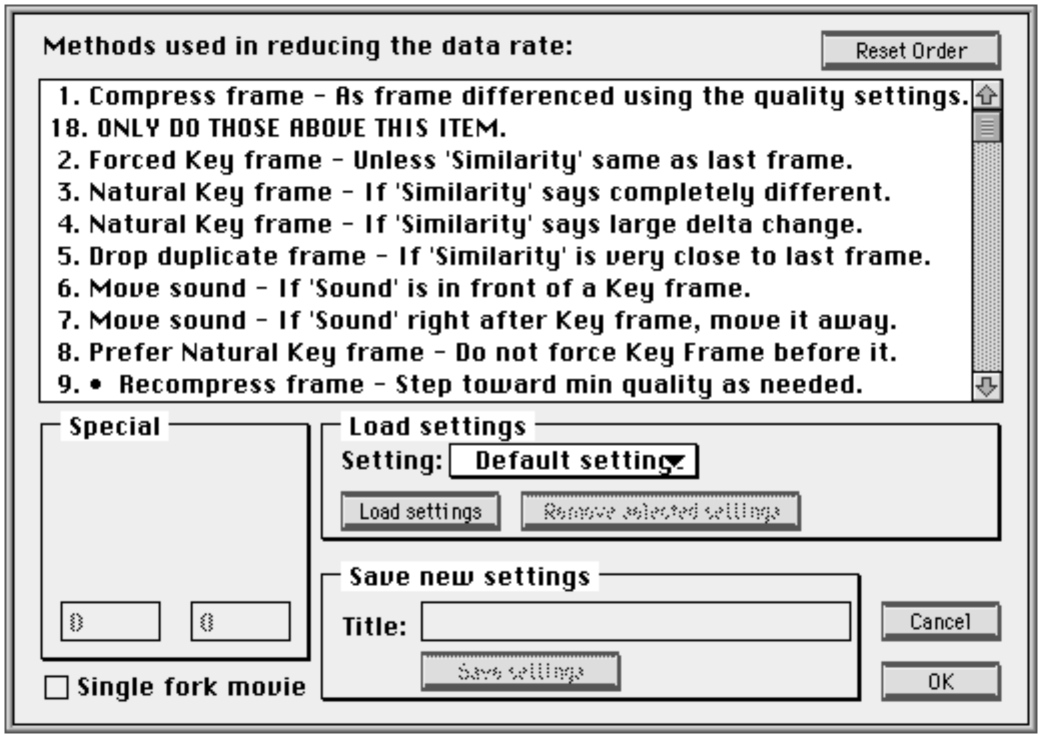
Crop by:

Left: 2
Right: 2
Top: 2
Bottom: 2

```
L = 2      T = 2
R = 238    B = 178
```

Figure 6
MovieShop's Cropping Dialog

DISABLE EXTRA COMPRESSION METHODS
MovieShop can apply a large number of techniques when compressing a movie to get the data rate you specify. These techniques are used by the Apple Compact Video compressor, but since the compressor itself takes care of all data rate limiting, all of these methods should be turned off. To turn them off, choose Methods from the Preferences menu. Then drag item 18 to the second position in the list of methods, as shown in Figure 7.


Methods used in reducing the data rate:

1. Compress frame - As frame differenced using the quality settings.
18. ONLY DO THOSE ABOVE THIS ITEM.
2. Forced Key frame - Unless 'Similarity' same as last frame.
3. Natural Key frame - If 'Similarity' says completely different.
4. Natural Key frame - If 'Similarity' says large delta change.
5. Drop duplicate frame - If 'Similarity' is very close to last frame.
6. Move sound - If 'Sound' is in front of a Key frame.
7. Move sound - If 'Sound' right after Key frame, move it away.
8. Prefer Natural Key frame - Do not force Key Frame before it.
9. • Recompress frame - Step toward min quality as needed.

Setting: Default setting

Special: 0 0

☐ Single fork movie

Title:

Figure 7
MovieShop's Methods Dialog

The version of MovieShop on the CD (v. 1.0c2) has a bug that causes it to use the settings for methods 2, 3, 4, and 5 even when they're below item 18. If you're working with that version, you should additionally set those methods to the following values to really disable them:

2. Forced Key frame — 255
3. Natural Key frame — 1
4. Natural Key frame — 200
5. Drop duplicate frame — 255

Again, for details about what these magic numbers mean, see the MovieShop documentation.

Once you've set up MovieShop this way, click "Make the movie," choose an output file, and sit back and watch the show. It can be a long wait, but the results will be worth it.

TIPS ON PLAYBACK

If you're developing an application to play QuickTime movies, there are four things you can do to make movie playback everything you'd hoped it would be:

- Optimize the movie's screen position and depth.
- Avoid clipping any portion of the movie.
- Hide the movie controller.
- Don't call WaitNextEvent as often.

OPTIMIZE SCREEN POSITION AND DEPTH

The position of the movie on the screen can affect playback performance. For greatest efficiency, the left edge of the movie should be aligned to a long-word boundary in video memory. A new function in QuickTime 1.5 called AlignWindow moves a window to the optimal screen location for movie playback.

The screen depth also affects playback. If the screen is set to millions of colors, the movie will play back more slowly than at 256 colors, because there are more bytes to move to the screen every frame. The Apple Compact Video compressor is optimized for thousands of colors (16 bits/pixel), so you'll get the best performance and quality at that depth.

AVOID CLIPPING

If any portion of the movie is clipped, playback performance will be substantially decreased because QuickTime will have to do a lot more work to draw the frames. Try to avoid overlapping windows and drawing to multiple screens. Be aware of the menu bar and the rounded corners on the edges of the Macintosh screen. Set the clipping region correctly for the movie.

HIDE THE MOVIE CONTROLLER

Displaying and updating the movie controller can cause the movie to play back more slowly, especially on low-end machines and for shorter movies, so you may want to hide it to achieve better playback. Still, it's desirable to have an option to show/hide the controller in your human interface. You might consider using the badge option for the movie controller to achieve this.

For more information on AlignWindow and other alignment routines, see the Image Compression Manager chapter of the QuickTime 1.5 Developer's Kit documentation, and *Inside Macintosh: QuickTime.*•

DON'T CALL WAITNEXTEVENT AS OFTEN
Since QuickTime performs all its drawing operations at main event loop time, the more often you call MoviesTask the better movie playback you'll get. However, most applications call WaitNextEvent once every event loop, which can go away for a fairly long time under System 7, effectively reducing the number of times MoviesTask gets called each second. To improve this, simply call WaitNextEvent only once a second or so while movies are playing. This will allow background tasks time to run but won't interfere with foreground event processing.

SEE YOU AT THE MOVIES!

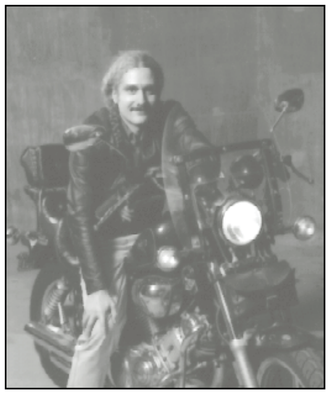
Now you know a little bit more about making and playing movies than you did before you sat down with this article. You understand what the tradeoffs are in making CD-playable movies: to get smooth playback, you may need to limit the frame size and rate, minimize the differences between frames, and trade off audio for video quality. You know that to get the best possible quality when you capture, you need to start with a clean source, adjust levels, and grab at a larger size than you need. You know how to use MovieShop with the Apple Compact Video compressor to compress your movie. And you know that to get the best possible playback, you need to optimize the movie's screen position and depth, avoid clipping any portion of the movie, hide the movie controller, and not call WaitNextEvent as often.

In a nutshell, if you want your movie to play well on a Macintosh LC II with an AppleCD SC and thousands of colors, you should use the following setup:

- Frame size: 240 x 180
- Frame rate: 12 fps
- Sound rate: 11.127 kHz
- Data rate: 100 kilobytes/second
- Compressor: Apple Compact Video
- Key frame: every 12 frames

With this knowledge, you can really put this technology to work and produce a QuickTime movie with the best of them. While you may not become the next Steven Spielberg, you can at least see your name in lights (check out the *QuickTime 1.5 Developer CD* About Box movies to see the QuickTime team in action). See you at the movies!

THANKS TO OUR TECHNICAL REVIEWERS
Dean Blackketter, Bill Guschwan, Peter Hoddie, Eric Hoffert•



BE OUR GUEST

SYSTEM ENABLERS

C. K. HAUN

System 7.1 introduced *system enablers* to Macintosh system software architecture. Since their introduction on some of Apple's new machines, there's been some confusion about what system enablers are for, and developers have expressed interest in writing and using them. This column will shed some light on the subject.

THE BAD NEWS

Even before beginning to describe them, we have to emphasize that system enablers are *system software*, designed and intended solely for the use of Apple Computer. Detailed descriptions of their design and use will not be released. Their functionality and implementation *will* change; any developers who try to decipher enablers and implement their own are warned that they will fail in future system releases. Do *not* write your own enabler or modify a current one!

Kinda harsh, huh? But we really mean it: the functionality of system enablers makes sense only for system software. Also, modifying or creating an enabler without fully understanding how one works could cause the enabler mechanism to fail silently. This could result in a machine that appears to be working correctly but does not have the full enabler functionality active, causing very hard-to-isolate crashes or other problems.

OK, SO WHAT ARE THEY?

System enablers (which were called "gibblies" in early documentation and system development releases) replace the release strategy that Apple used in the past for minor system changes needed for new hardware. The old strategy was to release a new version of the system software, such as version 6.0.8 or 7.0.1. With enablers, the differences in hardware no longer require a new system release, but instead each new machine has its own enabler (if necessary) to make the system work for that hardware.

This change was made for two reasons:

- Creating an enabler instead of a whole new system release reduces Apple's quality assurance and testing time. By creating an enabler, we're testing new code only on new machines; past machines aren't affected since the new enabler won't run on those machines. This also reduces your testing time as a third party, since you no longer have to install a new system release on all your older machines and test for compatibility; you know the system changes will affect only the newly released machines.
- Enablers reduce user confusion and unnecessary upgrades. In the past, every time a machine was released, with its corresponding new software release, users of older machines were unsure whether they needed to upgrade to the new system. Many assumed that since it was newer, it was better. While this has been true occasionally, it usually isn't (upgrading from 7.0 to 7.0.1 on a Macintosh IIcx, for example, gives the user nothing new).

Apple will continue to use system extensions or components when new functionality is being added across the product line, as with QuickTime and Macintosh Easy Open. Enablers just broaden the range of options for enhancing the user's environment.

HOW DO THEY WORK?

Nope, no cheating — we really won't be describing the internals of enablers. But here are some general rules about their behavior.

An enabler is essentially an extension to the System file; from a programmatic standpoint it *is* the System file.

C. K. HAUN works in Apple Developer Technical Support, where he is the perennial winner of the coveted "Most Documentation Heaped on the Floor" award. Before coming to Apple he was a commercial developer, writing educational, game, and utility software on Macintosh, IBM, and Apple II platforms. His main focus in DTS is interapplication communication and application toolbox support. He's also single again; please see ad #298700 in the personal ads section of this issue.•

We welcome guest columns from readers who have something interesting or useful to say. Send your column idea or draft to AppleLink DEVELOP or to Caroline Rose at Apple Computer, Inc., 20525 Mariani Avenue, M/S 75-2B, Cupertino, CA 95014.•

GIBBLY HISTORY
BY CHRIS DEROSSI

When system enablers were being designed, it was clear they were going to be a mechanism for extending the system. Engineers began calling them INITs, since they served much the same purpose. This caused some confusion with people who thought we really were talking about INITs, and issues were raised about possible conflicts with other INITs, loading order, and so on. (These issues don't apply to system enablers at all.)

It was obvious to me that we needed to change the name to differentiate system enablers from other types of system extension mechanisms. We couldn't use the term *extension* because that was the public name for INITs. We couldn't use *addition* because that term already referred to things like QuickDraw GX and AOCE.

So, I decided to use a made-up word. I wanted a word that would be memorable and one that would sound funny when used in all seriousness by executives and upper management. Hence the name "gibblies."

The engineering documentation for gibblies states that the singular is "gibbly" and the plural is "gibblies." It also states that the "g" may be pronounced hard or soft depending on the speaker's preference. In practice, the hard form (as in "give") became the one used by those in the know.

There you have it. And, yes, it was very amusing to hear the made-up word being used all over the company in serious conversation.

The Resource Manager was changed slightly to recognize references to the System file (CurResFile(0), for example) as being references to the System file *and* the current enabler. The code or other resources included in the enabler file appear to actually reside in the System file.

Any new machine may have a system enabler. The enabler will contain the system-level code necessary to implement changes required for that machine. A single enabler may be used for a family of machines, or a separate enabler may be created for each new machine.

If there are multiple enablers in a machine's System Folder, the system will use only *one* of them. The system software (System file plus enabler) is responsible for arbitrating which enabler is used on a specific machine. It looks at the machine type it's currently running on, the machines that the enabler supports, and some enabler-internal applicability flags. Note that this is how the decision is currently made; as enablers are used for more machines and in different situations, more variables may be added to the decision process.

ENABLERS AND REFERENCE RELEASES
A specific enabler may not stay around forever. Apple has announced its intention to have "reference releases" every year to 18 months; these are the system upgrades that all users will be encouraged to install. Some enablers may be rolled into a reference release, so a machine that needed an enabler for System 7.1 may not need one for System 8. This is not a hard and fast rule; some enablers may stay around forever, depending on the functionality they enable.

THE BOTTOM LINE
System enablers make everyone's life easier by encapsulating system changes for new machines in one place. But they are not for non-Apple use, and developers cannot implement them with any hope of long-term success. The safest path to take is not to think of enablers as separate files. An enabler *is* the system; when you encounter one on a machine, you're looking at the System file. The traditional methods (INITs, cdevs, components, and so on) are still the only supported ways for developers to extend system functionality.

Thanks to Chris Derossi, Greg Marriott, and Dean Yu for reviewing this column.•

MACINTOSH Q & A

Q *When a request for information is passed to me through an Apple event, the direct object parameter of my reply event is a descriptor list that includes an AERecord of my information. When I use AEPutPtr and the AEPutParamDesc, is the data copied or merely referenced? Should I be disposing of the AERecord and/or the descriptor list, or should I expect AEProcessAppleEvent to dispose of them?*

A Whenever you make one of the AEPutXXXX calls, the Apple Event Manager *copies* the data you put into the list, event, or record, so as soon as you make the call you can dispose of the data you put. Thus, the following is correct:

```
AEPutParamDesc(&theEvent, keyDirectObject, &theSpec);
AEDisposeDesc(&theSpec);
```

And so is this:

```
HLock(myTextHandle);
AEPutParamPtr(&theEvent, keyDirectObject, typeText,
   (Ptr)*myTextHandle, GetHandleSize(myTextHandle));
DisposeHandle(myTextHandle);
```

The *only* two descriptors disposed of by the Apple Event Manager itself (at the conclusion of AEProcessAppleEvent) are the original Apple event and the reply Apple event. So, anything that you create and manipulate yourself should be disposed of by you when you add it to another Apple event record or when you're done with it. The two that you don't dispose of yourself are theEvent and reply, which are passed to you, as in:

```
pascal OSErr AEXXXHandler(AppleEvent *theEvent, AppleEvent *reply,
   long refIn)
```

This even holds true for AESend. When you send an event, you can immediately dispose of your copy of the event, as follows:

```
AESend(&myEvent, nil, kAENoReply, kAENormalPriority,
   kAEDefaultTimeout, nil, nil);
AEDisposeDesc(&myEvent);
```

Q *The System 7.1 Digest has a disturbing comment about GetMHandle — namely that it was never supported and will no longer work. Is this true?*

A This warning is misleading and is being corrected in future release notes. It applies *only* to pop-up menus created with the pop-up menu control. Before System 7.1, after a control was created GetMHandle would return the menu handle for the control, although it was never documented as doing so. In

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 in Apple's Developer Support Center; our thanks to all. Special thanks to Pete ("Luke") Alexander, Mark Baumwell, Brian Bechtel, Cameron Birse, Joel Cannon, Matt Deatherage, Tim Dierks, Steve Falkenburg, Nitin Ganatra, Bill Guschwan, C. K. Haun, Dave Hersey, Scott Kuechle, Edgar Lee, Jim Luther, Joseph Maurer, Kevin Mellander, Martin Minow, Ed Navarrete, Guillermo Ortiz, Dave Radcliffe, Brigham Stevens, Dan Strnad, Forrest Tanaka, and John Wang for the material in this Q & A column.•

System 7.1 it was changed so that the menu would be inserted into the menu list only when the control was getting ready to pop up the menu and deleted as soon as the control was done with it, so you could no longer use GetMHandle to retrieve the menu handle. The proper way to get the menu handle is from the mHandle field of the popupPrivateData structure. The handle to this structure is in the contrlData field of the pop-up menu's control record.

A corollary is that the pop-up control has always checked to see if the menu is already in the menu list. If it is, the control doesn't get the menu from the menu resource and doesn't delete the menu when it's done. You can use this feature, for example, if you want to create a menu with NewMenu rather than getting it from a resource. In this case, and all other cases where the application inserts and deletes the pop-up menu in the menu list itself, GetMHandle can be used to retrieve the menu handle because it's under the control of the application.

Q *I read that Photo CD discs can be read by the AppleCD SC Plus and the AppleCD 150. Does this mean a plain vanilla AppleCD SC can't read them? Is this a hardware limitation, or will there be a software fix?*

A Apple erroneously reported that the original AppleCD SC could not read single-session Photo CD discs. This turns out not to be the case; all of Apple's CD-ROM drives can read single-session Photo CD discs.

Two levels of support are available for Kodak Photo CDs: the ability to read the first session on the Photo CD itself, and the ability to deal with more than just the initial session of a multisession CD. The AppleCD 300i is the first CD-ROM player from Apple to support multisession Photo CDs. For details about both support levels, check the Tech Info Library on AppleLink.

Q *Inside Macintosh Volume V, page 103, says that when a PICT pattern opcode (for instance, 0x0012) comes along, and the pattern isn't a dither pattern, the full pixMap data follows the old-style 8-byte pattern. The pixMap data structure shown on page 104 starts with an unused long (baseAddr placeholder), followed by the rowBytes, bounds, and so on. However, looking at the Pict.r file on the October 1992 Developer CD, at the same opcode (BkPixPat == 0x0012), the first data field after the old-style pattern (hex string[8]) is the rowBytes field (broken down into three bitstrings). The baseAddr placeholder field isn't there. Which is correct?*

A The *Inside Macintosh* Volume V documentation on pages 103–104 is wrong. The Pict.r file correctly describes the format of the PnPixPat and BkPixPat opcodes. So there shouldn't be a baseAddr field in the pixMap record of a pattern as stored in the PnPixPat of a PICT. However, the baseAddr does occur in a 'ppat' resource as described on page 79. Thanks for pointing out this discrepancy.

Q *How do I find the correct time values to pass to GetMoviePict, to get all the sequential frames of a QuickTime movie?*

A The best way to find the correct time to pass to get movie frames is to call the GetMovieNextInterestingTime routine repeatedly. Note that the first time you call GetMovieNextInterestingTime its flags parameter should have a value of nextTimeMediaSample + nextTimeEdgeOK to get the first frame. For subsequent calls the value of flags should be nextTimeMediaSample. Also, the whichMediaTypes parameter should include only tracks with visual information, 'vide' and 'text'. Check the Movie Toolbox chapter of the QuickTime documentation for details about the GetMovieNextInterestingTime call. For a code example, see the revised SimpleInMovie on this issue's CD. The routine to look at is called DoGetMoviePicts in the file SimpleInPicts.c.

Q *My routine uses a dialog hook to set and retrieve certain values in new items added to the default box. Previously, with SFPPutFile, I was able to use a hit on the Save item to retrieve and save the values. This also works with CustomPutFile unless the Replace/Cancel dialog box appears, because the dialog hook routines are also called for it! With the dialog pointer now pointing at the small alert, any reference to expected items leads to disaster, since they don't exist. Isn't calling the dialog hook routine to respond to hits in the alert box wrong and therefore a bug?*

A Both Standard File and the Edition Manager in System 7 allow you to have control in your filter when one of the subdialog boxes comes up. You can differentiate between the main dialog and the subdialogs by looking in the refCon field of the dialog record passed to you. In Standard File's case, if the dialog is the main dialog, the refCon will be:

```
/* From StandardFile.h */
/* The refCon field of the dialog record during a modalfilter
   or dialoghook contains one of the following: */
#define sfMainDialogRefCon       'stdf'
#define sfNewFolderDialogRefCon  'nfdr'
#define sfReplaceDialogRefCon    'rplc'
#define sfStatWarnDialogRefCon   'stat'
#define sfLockWarnDialogRefCon   'lock'
#define sfErrorDialogRefCon      'err '
```

This is described in detail on page 26-18 of *Inside Macintosh* Volume VI, in the middle of the section that describes all the callbacks and pseudo items for Standard File under System 7. The main purpose of this is to allow your additional dialog items to react properly when another dialog box is brought up in front of them, not to allow you access to the subdialogs. Also, since Standard

File has no idea what types of items you've added to the dialogs, giving you control during subdialogs allows you to change the look of your subitems, or to keep them active if they need periodic time for any reason.

Q *How do I find the current KCHR resource?*

A Here's a method for getting a copy of the KCHR resource currently being used by the system. This method works for both System 6 and System 7.

```
{  long     keyScript, KCHRID;
   Handle   KCHRHdl;
   /* First get the current keyboard script. */
   keyScript = GetEnvirons(smKeyScript);
   /* Now get the KCHR resource ID for that script. */
   KCHRID = GetScript((short)keyScript, smScriptKeys);
   /* Finally, get your own copy of this KCHR. Now you can pass
      a proper KCHR pointer to KeyTrans. */
   KCHRHdl = GetResource('KCHR', KCHRID);
}
```

Q *When I use CopyBits to move a cGrafPort's portPixMap to another cGrafPort (my printing port), it works like a charm when background printing is turned on, but when CopyBits gets called with background printing turned off, the image that prints isn't the image at all. Why is this happening?*

A You should be aware that since you're copying the pixels directly from the screen, the baseAddr pointer for the screen's pixMap may be 32-bit addressed. In fact, with 32-Bit QuickDraw, this is the case. This in itself isn't a problem, since CopyBits knows enough to access the baseAddr of the port's pixMap in 32-bit mode, as follows:

```
mode = true32b;
SwapMMUMode(&mode);  // Make sure we're in 32-bit addressing mode.
// Access pixels directly; make no other system calls.
SwapMMUMode(&mode);  // Restore the current mode.
```

That's how you'd normally handle things if you were accessing the pixels directly yourself. Unfortunately, the LaserWriter driver doesn't know enough to do the SwapMMUMode and instead ends up copying garbage (from a 32-bit pointer stripped to a 24-bit pointer).

So why does background printing work? Because when you print in the background, everything is rolled into a PICT, which the driver saves off for PrintMonitor. Since the driver is using the standard QuickDraw picture bottlenecks to do this, and CopyBits knows to swap the MMU mode before copying the data into the picture, everything works great. Later, at PrintMonitor time, the picture is played back. Since the data is no longer 32-bit addressed, the LaserWriter driver doesn't have to call SwapMMUMode to do the right thing; it can just play the picture back.

The solution we propose for you is something similar. At print time (before your PrOpenPage call), call OpenPicture, copy the data from the screen with CopyBits, call ClosePicture, and then call DrawPicture within your PrOpenPage/PrClosePage loop. That should do the trick.

Note that copying bits directly from the screen is not something we recommend. Unless you have no alternative, you should *always* copy from the original source of the data instead.

Q *Is there a way to read Greenwich Mean Time offsets from the Map control panel?*

A There's actually a system-level call to find out where you are. It's a Script Manager call named ReadLocation (used by the Map control panel), which returns a structure giving you all the information you need. Here's a description of the call, copied from MPW 411:

```
pascal void ReadLocation(MachineLocation *loc)
   = {0x205F,0x203C,0x000C,0x00E4,0xA051};
File {CIncludes}script.h
```

In C:

```
pascal void ReadLocation(MachineLocation *loc);
pascal void WriteLocation(const MachineLocation *loc);
```

These routines access the stored geographic location and time zone information for the Macintosh from parameter RAM. For example, the time zone information can be used to derive the absolute time (GMT) that a document or mail message was created. With this information, when the document is received across time zones, the creation date and time are correct. Otherwise, documents can appear to be created after they're read. (For example, someone could create a message in Tokyo on Tuesday and send it to San Francisco, where it's received and read on Monday.) Geographic information can also be used by applications that require it.

If the MachineLocation has never been set, it should be <0,0,0>. The top byte of the gmtDelta should be masked off and preserved when writing: it's reserved for future extension. The gmtDelta is in seconds east of GMT; for example, San

Francisco is at minus 28,800 seconds (8 hours * 3600 seconds per hour). The latitude and longitude are in fractions of a great circle, giving them accuracy to within less than a foot, which should be sufficient for most purposes. For example, Fract values of 1.0 = 90°, -1.0 = -90°, and -2.0 = -180°. In C:

```
struct MachineLocation {
   Fract latitude;
   Fract longitude;
   union {
      char  dlsDelta;   /* signed byte; daylight savings delta */
      long  gmtDelta;   /* must mask - see documentation */
   }  gmtFlags;
};
```

The gmtDelta is really a 3-byte value, so you must take care to get and set it properly, as in the following C code examples:

```
long GetGmtDelta(MachineLocation myLocation)
{
   long  internalGMTDelta;
   internalGMTDelta = myLocation.gmtDelta & 0x00ffffff;
      if ( (internalGMTDelta >> 23) & 1 ) // need to sign extend
         internalGmtDelta = internalGmtDelta | 0xff000000;
      return (internalGmtDelta);
}

void SetGmtDelta(MachineLocation *myLocation, long myGmtDelta)
{
   char  tempSignedByte;
   tempSignedByte = myLocation->dlsDelta;
   myLocation->gmtDelta = myGmtDelta;
   myLocation->dlsDelta = tempSignedByte;
}
```

Q *Did you hear that they had computer music at Clinton's inauguration?*

A Yes, they danced to Al Gore Rhythms.

Q *What (if at all) is the limitation on the number of files in a folder? In other words, is there a number N, such that if I have N files in a folder, and I try to Create file number N+1, I'll get some error?*

A In general, the number of files that can be put in an HFS directory is unlimited; there isn't any point at which you'll receive an error from Create, because new file description records can almost always be created. The only way you can get a disk full error back from Create is if the catalog file needs to grow to add your new record and the disk is full, but this should be extremely rare; even when the disk is full, there's generally room to create dozens of files or folders before the catalog file will need to be enlarged, as it's grown in relatively large chunks.

There are, however, a couple of related limits on large numbers of files. Because HFS allocates space in "allocation blocks," and there can be at most 65,536 allocation blocks on a volume, there's a limit of 65,536 files that contain data on a volume. If your disk is full with 65,536 one-block files, you'll probably be able to create more files (Create will succeed), but no data will be able to be written to them. In reality, the limit on the number of files is somewhat smaller; the catalog and extents files will each occupy space. Also, because the allocation block size needs to be an even multiple of 512 bytes, most volumes won't have a full 65,536 allocation blocks; they'll have as many as they can, somewhere between 32,767 and 65,536 (except for small volumes, which may have less). In addition, each fork (either the data fork or the resource fork) of a file needs to be separately allocated, so each counts as a file for this calculation.

There's also a practical limit on how many files can be placed in a folder. HFS can maintain as many files as required in a directory; however, because the index field is a short, if there are more than 32,767 files in a folder, they can't be enumerated. Thus, while they can be identified and opened by name, there's no way to index through them to determine what's in the directory without deleting or moving some of the files.

These limit descriptions apply to HFS only; other file systems may be available on the Macintosh, such as MFS, MS-DOS, ISO 9660, or virtually any file system via an AppleTalk Filing Protocol (AFP) translator. These descriptions also don't include limitations of the Finder, the Standard File Package, or any other intermediaries. The Finder and Standard File are likely to become quite unusable long before you run into the limits of HFS. Also, while HFS will continue to function, it wasn't designed for optimum performance with extremely large numbers of files; for more dire warnings on this subject, see the Macintosh (Overview) Technical Note "Don't Abuse the Managers" (formerly #203).

Q *In the two-byte script version of our application, we need to insert certain characters such as "-" and "%" into some of our strings. How can we do this, since these are obviously only one character long in C?*

A All 7-bit ASCII characters (codes less than 127) are maintained as such in all two-byte scripts. If your routines just concatenate existing (localized) strings and characters, you don't have to worry about anything. Otherwise, you'll need to call CharByte (*Inside Macintosh* Volume V, page 306, and Volume VI, pages 14-45, 14-114, and 14-124) while walking the bytes in the string.

In the Macintosh (Text) Technical Note "Fond of FONDs," the part about OutlineMetrics was updated recently to be "two-byte aware." The code fragment there should help you with strings containing text for two-byte scripts. See also the article "Writing Localizable Applications" in this issue.

Q *The Icon Utilities routine GetIconCacheData leaves two bytes of garbage on the stack. This surfaced as a problem for me because it prevented a saved register from getting restored properly. SetIconCacheData probably has the same problem, since it calls the same trap internally. I solved the problem by encapsulating GetIconCacheData within my own static function, like so:*

```
static OSErr _GetIconCacheData( Handle theCache, void **theData )
{
return   GetIconCacheData( theCache, theData );
}
#define  GetIconCacheData  _GetIconCacheData
```

I then call GetIconCacheData normally. The #define redirects my call to my static wrapper function. The extra two bytes on the stack are recovered when the wrapper function UNLKs and returns. This method has the advantage that the calling code will still work even after the trap is fixed. Am I correct?

A You're quite correct; this is a bug. Here's the offending code from the source:

```
EXIT     MOVEA.L     (SP)+, A0   ; Pop return address into A0
         ADDQ.L      #6, SP      ; Point stack at return value
         MOVE.W      D0, (SP)    ; Put return value on the stack
         JMP         (A0)        ; Return
```

As you can see, the exit routine is adding 6 to the stack to clear up the input parameters instead of 8 (handle and handle), so an extra word of garbage is being left on the stack. Thanks for letting us know about the problem.

Q *When I double-click a document that launches my application, the current directory for the Standard File Package (at location $398 in memory) is set to the directory of my application and not my document. This seems to be a bug according to the text on page 3-31 of the new Inside Macintosh: Files manual. Is there anything special I have to do?*

A You're right. The behavior described in *Inside Macintosh: Files* isn't entirely correct. It should say that the first time your application calls one of the

Standard File Package routines, the default current directory (that is, the directory whose contents are listed in the dialog box) is determined by the way your application was launched.

- If the user launched your application directly (perhaps by double-clicking its icon in the Finder), the default directory is the directory in which your application is located.
- If the user launched your application indirectly (perhaps by double-clicking one of your application's document icons) and your application is passed Finder information, the default directory is the directory of the last document listed in the Finder information. The Finder information is the data referenced by AppParmHandle and accessed by the Segment Loader routines CountAppFiles, GetAppFiles, ClrAppFiles, and GetAppParms.

Note that applications that are high-level event aware are passed the list of documents to open or print in a kAEOpenDocument or kAEPrintDocument Apple event. There's no Finder information (AppParmHandle will be NIL) and the default directory is the directory in which your application is located.

Q *Sometimes, at the beginning of a PBRead on a serial port, I get back a result code of -90 in the completion routine. I don't quite know how to handle this error, because I can't find a -90 result code anywhere. Any idea what -90 means?*

A According to the MPW Errors.h interface file, -90 is a BreakRecd result. (The interface files are always a good place to look for error codes and calls that you can't find.) The serial driver returns that error to a pending Read if the SCC chip detects a break condition.

Q *Why does MacApp use the Initialize and Free methods instead of the normal C++ constructor and destructor methods?*

A MacApp doesn't use constructors for historical reasons. Object Pascal was used in MacApp 2.0.1, which doesn't have constructors and destructors as part of the language, and as a result these facilities had to be provided as part of MacApp instead of the language.

MacApp 3.0 designers tried to achieve backward compatibility with applications written with older versions of MacApp based on Object Pascal. Because of this, the designers decided to stay with the Initialize and Free functions rather than just have an instance of the object declared and destroyed with **new** and **free**.

Q *Can I use the CompressPicture routine to spool in a source picture from disk by overriding the QuickDraw proc getPicProc as documented in Inside Macintosh Volume*

V, pages 88-89? I'm trying to save the contents of an off-screen GWorld as a compressed PICT resource. Unfortunately there's no direct way to compress the GWorld's pixel map to a resource.

A We definitely don't recommend trying to spool in or out the results of CompressPicture or CompressImage. We recommend doing one of the following instead:

- You can compress the GWorld using CompressImage and then call OpenPicture, DecompressImage, and ClosePicture using a data-unloading picture proc. The drawback here is that you need to have a copy of the compressed image in memory.
- If it's unacceptable to have an entire compressed image in memory, you can consider banding along with data unloading: Call OpenPicture, then CompressImage and DecompressImage on a band, CompressImage and DecompressImage on another band, and so on. When all bands are done, call ClosePicture. The drawback for this is that the compressed picture will have bands of image data that won't display well dithered. This could be an issue, but the best way to find out is to try it.

The second suggestion is probably the best idea if you want to keep your memory footprint small. But much of the decision depends on your application.

Q *Our product's sound/video synchronization is way off with QuickTime 1.5; it worked perfectly with QuickTime 1.0. The video can't keep up with the sound, especially on the full-screen movies. The movies are playing much slower with 1.5. Isn't QuickTime 1.5 supposed to make movies play faster?*

A Your movies probably aren't properly interleaved. When you add sound to a movie with SoundToMovie, the sound is added to the end of the video data. We recommend that sound and video be interleaved so that the hard drive doesn't have to spend extra time seeking between media that store video and media that store audio on a hard drive. The data handler prefers to be able to sequentially read through a movie file. This is especially important for slower Macintosh models that don't have the extra CPU and SCSI access speed to spare.

QuickTime can accommodate for some noninterleaved data by caching an entire sound track of a movie if small enough. However, the size of a cache is internal to QuickTime and can't be depended on. It's possible that different QuickTime versions could have different cache sizes since we've been recommending that video and sound movies be interleaved. The result could be that the extra disk-seeking time has caused sound and video to be out of sync for slower machines such as the Macintosh LC.

One way to check interleaving is to resave the movie using Movie Converter (or some other program that flattens movies) in a flattened format. Movie Converter uses QuickTime's FlattenMovie call to do this. The steps Movie Converter takes are: choose Save As, select "Make movie self-contained," and save to a new movie file. This new movie should play back with correct video and sound sync.

You can actually see the problem if you examine the movie with MovieShop, a program that lets you deal with QuickTime movies at a movie data level. For example, if you select the Play information button that's in the window after you open a movie, the program will display a time graph showing you where the video and sound data are saved in the continuous data stream. If the movie is interleaved, the green (for video) and red (for sound) indicators are interleaved. If the movie isn't interleaved, the green indicators are clumped together in the beginning of the file, and the red indicators (for sound) are at the end.

The latest version of MovieShop and documentation is available on this issue's CD. MovieShop and related information are discussed in the article "Making Better QuickTime Movies" in this issue.

Q *How long can a Macintosh filename be on an international system? Our program currently assumes a maximum filename length of 31. What does the length byte of a Pascal string signify on an international system — the actual length in bytes of the string, the logical length (the number of international characters), or neither?*

A The Macintosh file systems — the flat (MFS) and the hierarchical (HFS) — have (reasonable) limitations on the length of filenames. The limits on the lengths refer to the number of *bytes* required for the strings. Usually, filenames are represented as Pascal strings. The first byte of a Pascal string indicates the number of bytes occupied by the string. In the worst case in a two-byte script, only 15 two-byte characters fit into a Str31. Compared to one-byte scripts, this isn't so bad, however: two-byte characters tend to carry much more meaning than two of our familiar one-byte characters!

Q *What is QuickTime for Windows and what hardware do you need to run it?*

A QuickTime for Windows makes QuickTime a cross-platform technology that you can use in both the Macintosh and the Microsoft Windows programs that you're developing. With it, you can create multimedia programs on the Macintosh and be able to add the playback of QuickTime movies to any PC application running Microsoft Windows 3.1.

The *minimum* hardware configuration required for QuickTime for Windows is the following:

- A personal computer with an 80386SX or greater CPU
- A CPU speed of 20 MHz or higher
- 4 MB of conventional and extended memory
- A hard disk with at least 4 MB free for the basic QuickTime for Windows software
- A graphics adapter and mouse (or other pointing device) supported by Microsoft Windows

Optional hardware:

- A CD-ROM drive supported by Microsoft Windows (if installing from a CD)
- A sound card supported by Microsoft Windows
- Additional free disk space if you want to keep movies and pictures on your hard disk

The software requirement for running QuickTime for Windows is Microsoft Windows 3.1. To write QuickTime for Windows programs, you must have (in addition to the QuickTime for Windows Developer's Kit) a development package such as Borland C++ 3.1 or Microsoft C/C++ 7.0.

The sample QuickTime for Windows executables may be played under Windows 3.1 immediately following QuickTime for Windows installation and configuration, without C or C++ being present.

Q *I tried to unmount a volume shared with Macintosh File Sharing from my program as follows: I shut down the file service with the SCShutDown server control call; I call SCPollServer to make sure the file service is really off (scServerState = SCPSJustDisabled); then I call PBUnmountVol to attempt to unmount the volume. It didn't work because PBUnmountVol fails with fBsyErr (-47). I broke on the _UnmountVol trap because the AppleShare PDS file, where the file server keeps the access privilege and share-point information for the shared volume, was open. Why is AppleShare PDS still open when I've turned the file service off? How can I close it and unmount the volume?*

A SCPollServer returns the state of the file service, not the file server application (in this case, File Sharing Extension is the file server application). When SCPollServer returns a server state of SCPSJustDisabled, the file service is off; however, the file server application may or may not still be running. The AppleShare PDS file will eventually get closed before the file server application quits.

There's an easy way to determine when the File Sharing Extension application has quit (and thus when the AppleShare PDS file is closed): just use the Process Manager GetNextProcess and GetProcessInformation calls to find out when File Sharing Extension is no longer running. The File Sharing Extension application has a processType of 'INIT' and a processSignature of 'hhgg'. Here's a function you can use to see if File Sharing Extension is running:

```
FUNCTION FileSharingAppIsRunning: Boolean;
   CONST
      FileSharingSignature = 'hhgg';   {Macintosh File Sharing}
   VAR
      err:        OSErr;
      myPSN:      ProcessSerialNumber;
      myPInfoRec: ProcessInfoRec;

BEGIN
   myPSN.highLongOfPSN := 0;  {Start at beginning of process list}
   myPSN.lowLongOfPSN := kNoProcess;
   myPInfoRec.processInfoLength := sizeOf(ProcessInfoRec);
   myPInfoRec.processName := NIL;    {Don't need process name}
   myPInfoRec.processAppSpec := NIL; {Don't need process location}
   FileSharingAppIsRunning := FALSE; {Haven't found it yet}
   WHILE (GetNextProcess(myPSN) = noErr) DO
      IF GetProcessInformation(myPSN, myPInfoRec) = noErr THEN
         IF (myPInfoRec.processSignature = FileSharingSignature) THEN
            FileSharingAppIsRunning := TRUE; {Found it}
END;
```

After shutting down the file service, your event loop will need to poll with FileSharingAppIsRunning because you must give the file server application processing time to close files, dispose of memory, and perform other shutdown operations. If you poll with FileSharingAppIsRunning without giving other processes time, File Sharing Extension will never shut down.

Q *I'm having trouble understanding the problems of dealing with potentially infinite loops in interconnected, distributed applications. Can you help me?*

A Please see the next message.

Q *I'm having trouble understanding the problems of dealing with potentially infinite loops in interconnected, distributed applications. Can you help me?*

A Please see the previous message.

Have more questions? Need more answers? Take a look at the Macintosh Q&A Technical Notes on this issue's CD and in the Dev Tech Answers library on AppleLink.•

DAVE JOHNSON

THE VETERAN NEOPHYTE

TINY FUTURES

I recently attended the First General Conference on Nanotechnology. The conference was sponsored primarily by the Foresight Institute, an organization based in Palo Alto, California, whose sole self-stated goal is to disseminate information about nanotechnology, to inform the public about the topic, and to just sort of do whatever seems necessary to get society ready, to prepare the ground, for the advent of this world-transforming future technology. (Apple also helped sponsor the conference, a fact I didn't even know until I arrived.) There had been previous "research" gatherings dealing with nanotechnology, but those were primarily for scientists and other gearhead types (most of whom, it should be noted, showed up for this one, too). This, though, was the first such gathering intended for the general public, and the first intended to foster open discussion on the topic by all kinds of people.

It was absolutely fascinating, on lots of levels, and a total blast! It was intellectually and scientifically stimulating, of course, and that's a lot of fun by itself, in a quiet sort of way. But it was also an unparalleled opportunity to *watch some really weird people*, and that's fun in a much larger, noisier sort of way. Watching wild ideas being bandied about by wild, zealous people is fine sport, and this was the perfect place for it. The enthusiasm among the participants was sizzling, and the whole thing smelled sort of cultish, almost religious in its zeal and drive. As I said, a blast!

I don't want to imply that it was a circus, that the participants were doddering, babbling boobs or mindless, frenzied fanatics. Nothing could be further from the truth. On the contrary, there were *all* kinds of people there — writers, computer folks, venture capitalists, cryonicists, physicists, doctors, marketers, biologists, businesspeople, you name it — and the ideas being discussed were often taken very seriously. But there *was* a healthy contingent of fringe-dwellers and edge-runners, people whose beliefs often set them slightly apart from your average, ordinary citizen (whatever that is). People who like to peek over the edge — any edge — to see what's there. Nanotechnology is a precipitous edge indeed.

Nanotechnology has been getting a lot of press lately, but for those of you who haven't heard or read about it yet, let me run you through the basics. Nanotechnology is the brainchild of one Dr. K. Eric Drexler, MIT graduate and technological visionary extraordinaire. (Some of the ideas and concepts of nanotechnology weren't new, but Drexler brought them together, gave them a name, and carried them much further than anyone else had dared.) Simply put, nanotechnology is the ability to precisely and completely control the structure of matter at the molecular or atomic level, by building the desired substance or structure molecule by molecule or atom by atom, placing each in the precise location we want. This capability has implications and ramifications without end, as we'll soon see. But the central spark of the idea — placing atoms one by one — remains simple and elegant, and we should be careful to remember that fact as we wander, often lost in the churning chaos, through the landscapes of possibilities that this oh-so-simple idea can generate.

(It's equally important, by the way, to remember that nanotechnology is still a fairy tale, though possibly a prophetic one. All the books, meetings, articles, discussions, and press coverage are about something that can't be done, at least not yet. So although the general buzz around the conference tended to use the present tense — an indication of the rampant confidence most of these people possess — it's all still

DAVE JOHNSON and his brother Doug decided to dig to China one fine morning long ago, but it turned out to be harder than they thought. The digging slowed and finally stopped around lunch time, both of them exhausted and hungry after digging perhaps 14 inches. They decided to finish the next day. That night, falling asleep, Dave decided to dive through head first, so he'd be right side up when he got to the other side. Dave still hasn't been to China.•

a dream, though as compelling and disturbing a dream as any I've ever known, and one that's being dreamt by some very, very capable minds.)

The term *nanotechnology* has been popularized lately, for better or for worse, and has been applied to a number of very different technologies that are decidedly *not* what Drexler has in mind. Their only similarity with Drexler's nanotechnology is that they involve very small scales. You've probably all seen those electron microscope pictures of little bitty gears and shafts and motors and flaps that have been carved from silicon. There's a large effort under way to build these micromachines, and it's a fascinating technology, but it's not what we're talking about. Those efforts, like many others that are popularly called nanotechnology, are characterized by starting with some block of material — a bazillion atoms in a chunk — and carving out bits or adding bits like a sculptor to get the shape you want. The resulting parts are still made of a bazillion atoms; you're still dealing with *clumps* of atoms at a time. Drexler calls this *bulk technology*. Admittedly, the clumps are getting very, very small these days, but it's really just a refinement of the same manufacturing technology we've had since the stone age: take a hunk of material and shape it.

Drexler's nanotechnology — he promoted the term *molecular manufacturing* at the conference, and it's a more descriptive, if less poetic, name — comes around from the other side. It starts with individual atoms and molecules and puts them together, essentially one piece at a time, to build up the desired material or structure. This is the key distinction between bulk technology and what we're calling nanotechnology.

Drexler also postulates a nanomachine he calls an *assembler*. It's a general-purpose atom positioning machine, sort of a nano-scale robot, complete with sensors to detect the atom or molecule, some sort of "gripper" to hold and position it, and a powerful computer to control the thing. This is the little bugger that really cuts nanotechnology loose. If you have assemblers, you have the proverbial general manufacturing machine: you can build anything, including more general manufacturing machines. (Actually, these days people talk more about "mills" than individual robot-like assemblers. Mills are like production lines, with conveyor belts rather than arms, and a continuous flow of material. Although in this column I'll use the word *assembler* what I really mean is this: a machine that can arrange atoms and molecules in a general way, and therefore can be used to build anything we can think of.)

OK, so now that we have a handle on what we're talking about, let's play that game we all love so much, Predicting the Future. If we *could* be atomic bricklayers, if we *could* command the structure of matter, just what sorts of strange things would we build? Here's where we can really have some fun. The ramifications of being able to build things atom by atom are of course myriad. Nanotechnology is one of those ideas that, when planted firmly in human minds, seems to serve as a catalyst, breeding innumerable possible future scenarios. It's a technology (a nonexistent one, remember!) that could conceivably touch and transform every important aspect of our lives.

Many scenarios are immediately apparent. For one thing, molecular manufacturing promises materials that are lighter, stronger, cheaper, and just generally better in every way. As any materials scientist will be happy to tell you, materials in the real world are generally riddled with defects. Carving away at them or adding other bulky bits to them as we do today doesn't change that fact, and actually often exacerbates it. But if we could build up our materials atom by atom, each precisely placed, the resultant material would be atomically precise, atomically *perfect!* This would mean that we could build much, much lighter weight structures, using lots less material to do the same job. That in itself has enormous benefits, but it's just the beginning: beyond improving existing materials, we could build any *new* material we can think of, as long as it's allowed by the laws of nature.

And the manufacturing processes themselves could be made amazingly efficient, using cheap and plentiful raw materials, producing virtually no waste, and consuming very little energy, if any at all. At the conference there was much talk, only partly tongue in cheek, of a tabletop nanofabricator, about the size of a microwave oven, with four rubber feet and a fan in the back, plugged into the wall. According to Drexler's calculations we could feed this thing 1.6 kilograms per hour of feed stock solution (acetone, I think he said: a cheap and plentiful source of carbon and hydrogen) and 0.8 kg/hr of atmospheric oxygen, and out the other end would come 1.0 kg/hr of diamond (or whatever carbon material you have in mind), 1.5 kg/hr of chemically pure water, 1.1 kW of waste heat, and, as a by-product of the process, 3.6 kW of surplus electricity. (Why plug it in? So that you can deliver the electricity to the power grid.)

With manufacturing processes like that, economics is suddenly turned on its ear. Some say there would be no more poverty, that since with assemblers we'd be able to make nearly anything for nearly nothing, precious materials would no longer be precious; we could make treasure from garbage! Some say we could make food, or better yet, create new materials and technologies that let us make full use of the food we already produce, bringing an end to world hunger.

Going even further out on an already shaky limb, let's examine some medical implementations. If we could build assemblers, we could build other nano-scale robots to do our bidding. Tiny observation machines could be injected into our bloodstream to seek out and report any damage, effectively giving doctors eyes into your body at the cellular level. How about nanogoop that you pour on a wound that disinfects it, seals it, and accelerates the regeneration of lost tissue? The cryonicists — people who have themselves frozen for some hopeful future awakening — have pinned most of their hopes for resuscitation on some sort of cell-repair machines that can go in and repair the tissue damage due to freezing. One speaker at the conference, an MD researching organ cryopreservation, planted his tongue firmly in his cheek and went truly wild with his speculations. Get this: subcutaneous "smart" armor that could see a blow or a bullet coming and *react*, bracing itself or maybe even pulling your skin away from the danger! He went on to talk about the "tradeoffs involved in becoming a flying person," a topic "no one has talked about before." (The conclusion was effectively that wings are very inconvenient and would really get in the way when you weren't flying, but they're probably worth the trouble.)

Then there are, of course, the "dark" scenarios. If it's possible to build assemblers, it will probably also be possible to build *dis*assemblers: imagine scavenger nanostuff whose programming has gone out of control, so that it disassembles anything it comes into contact with and just builds more copies of itself (I can see you artificial life fans pricking up your ears). This is known as the "gray goo" scenario. (Speculations about this sort of out-of-control goo cause equally energetic counter-speculations: encrypting the program so that a one-bit error turns it to hash; anti-goo goo — so-called "blue goo" — that recognizes and destroys the gray stuff; using a "broadcast architecture" so that the machines have no autonomy at all, and thus can't get out of control in the first place; and so on.)

Machines building copies of themselves opens up a Pandora's box of implications and problems: if these machines are the least bit autonomous and there's the possibility of mutation — of nonfatal errors when building new copies — they'll naturally begin to evolve! Drexler makes the good point that at the atomic level things are either exactly right or exactly wrong, that nanomachines are wrought in a fundamentally *digital* medium and are therefore brittle, so errors will tend to be catastrophic and bring things to a screeching halt. But it seems to me that the nanomachines we're talking about — machines that can sense their environment, harvest raw materials, and build copies of themselves, including their own instructions — are so complex that a digital error, a bit error, may *not* bring everything to a halt, and may indeed change the operation of the machine in subtle, mysterious ways. Anyone who has ever programmed a computer can testify to that.

You can also bet that if one group of people has this technology and another doesn't, and those groups don't like each other, the results could be many kinds of ugly.

Nanoweaponry could be more insidious and invisible than any biological weapon, more tenacious than radiation, and ultimately more destructive than any bomb.

Good or bad, many of these scenarios seem pretty far out there. Which brings up a very good question: will it really ever happen? Will we ever be able to build a general manufacturing machine? Can we ever gain that degree of control over matter? Naturally, Drexler thinks so, and so do many others. It's instructive to take a look at some of the practicalities involved.

First there's the problem of scale. Obviously, there are an awful lot of atoms in a piece of material that's, say, the size of your fist. Won't it take a long, long time to build up something that size atom by atom? Well, we can do the arithmetic to find out: Let's assume that we could put individual atoms together at a rate of 100 atoms per second. And let's say we want to build 12 grams of diamond. (Diamond is a very popular material in these nano-examples, not because of its worth and beauty but because of its amazing hardness and stability and the fact that it's made of carbon, a very common element.) If you've ever taken a chemistry course you know that 12 grams of carbon contains 6.02×10^{23} atoms (Avogadro's number, remember?). A hundred atoms per second is 864,000 per day, assuming no time off for the little buggers. This is about 3.15×10^9 atoms per year. So we could get our 12 grams of carbon in only, let's see, 6.02×10^{23} atoms divided by 3.15×10^9 atoms per year is roughly 2×10^{14} years. *That's two hundred million million years!* About fifty thousand times the estimated age of the earth! To build less than half an ounce of carbon. Hmm. Clearly we're going to have to do something drastic, numbers-wise, for this to be at all useful. Because of the extreme scales involved, the numbers quickly fly wildly out of control.

Well, the only way I've heard discussed to bring those numbers under our control is to fight back with large numbers of our own, in this case large numbers of nanomachines (or conveyor belts, or whatever) working simultaneously. And the only way to get that many machines is to build nanomachines that can reproduce, that can build copies of themselves. It's like the old story about the blacksmith shoeing a horse, who charged a penny for the first nail, two pennies for the second, four for the third, and so on: he became rich on one horse. Back to our example, we'd need about 1.0×10^{17} machines working simultaneously, each placing 100 atoms a second, to get our 12 grams in under 24 hours. How long will it take to make that many machines? Well, if each machine can build a copy of itself in one hour (a conservative estimate: an average bacterium does it faster), and we start with one machine, I come up with something like 56 hours. Very doable. And, of course, if we let it go for one more hour, we have twice what we need! When you've got geometric progression on your side, you've got a *friend!*

OK, so maybe we can handle the scale problem, as long as we're willing to let machines self-reproduce. What about the problem of actually reaching in there and grabbing atoms or molecules? Is it really possible to build molecular "hands"? Well, there are several technologies, each making rapid progress, that are converging on this capability. *Scanning Probe Microscopy* is a technology that shows great promise in positioning individual atoms (by now you've probably all seen that picture of "IBM" spelled out in individual atoms on a plate). Molecular biotechnology is another promising avenue: molecular biologists are gaining an amazing degree of control of the molecules of life, a degree of control that looks as though it will continue to increase quickly. (In a sense, using bacteria to manufacture insulin, as is done today, *is* molecular manufacturing; it's just that most of the nanomachinery was borrowed from living things, rather than designed from scratch.) The point is that there are many paths that may lead us to atomic control of matter, not just one.

But there are also some compelling arguments against the "full" vision, the future in which everyone has assemblers and we can make anything we want. First off, an assembler is a very, very complex machine, much more complex than anything humans have ever built, and I'm honestly not convinced that human beings are capable of *ever* deliberately building something like that. (Accidentally, maybe, but that's another story.) *You* try to design a machine, constructed entirely from sticky marbles, that can build a copy of itself from ambient sticky marbles floating nearby. Oh yeah, and it has to also be fully programmable, so that it can be instructed to build anything *else* (also from ambient sticky marbles). To my knowledge, no human has ever succeeded, at *any* scale, in building a purely mechanical machine that can build a copy of itself. But that's exactly what an assembler needs to be.

Even if we can get the mechanics together, there's still The Software Problem; complex software is always buggy, and the more complex it is, the further from "correct" it will be and the more unpredictable will be the results of such errors. Anyone who says otherwise doesn't know what they're talking about. The software to control a machine that can sense its environment, locate the appropriate parts, grab them, turn them the right way, and stick them to other parts is going to be more complex than anyone today knows how to write.

Then there's this sad fact: no technology is ever equally available to all people at its introduction, or for that matter for as long as there's some advantage, economic or otherwise, to maintaining control over it. And the advantages to maintaining control of this one are obviously huge. What if you're the first one on the block with that tabletop machine (four rubber feet, remember) cranking out intricately structured diamond struts and electricity? Are you going to stop building your struts and start building copies of your machine for everyone else, or would you be tempted to sell those excellent struts (that no one else can make yet) for just a little while first, and build up a nest egg? OK, I'll give *you* the benefit of the doubt, but what about that snake oil salesman over there? What do you think he would do? How about your government? If they had it first would they give a copy to you? To another country? These are hard questions, very hard indeed.

There is, of course, one piece of irrefutable proof that nanotechnology can ultimately work: life itself. In a very real way, we *are* nanotechnology. What are we but a mass of autonomously running nanomachines frantically making copies of themselves and each other? What are we but "out of control" nanostuff that has attained a very high level of organization? Some theorists believe that the odds of the emergence of life are better than previously thought, perhaps even that it's inevitable. They believe that matter has an inherent tendency to organize itself, and that we are the result of that tendency. (I get this creepy image of matter sort of turning around to look at itself.) If we gain total control over matter, perhaps we will also, as part of the bargain, gain total control over life. Now *that* would be something to write home about!

RECOMMENDED READING

- *Engines of Creation* by K. Eric Drexler (Doubleday, 1986). The first book detailing Drexler's ideas.
- *Unbounding the Future* by K. Eric Drexler and Chris Peterson, with Gayle Pergamit (Morrow, 1991). A popular account of the implications of nanotechnology.
- *Nanosystems* by K. Eric Drexler (Wiley Interscience, 1992). A nanoengineering textbook with detailed designs and calculations. The ultimate in theoretical engineering.
- *Blood Music* by Greg Bear (Arbor House, 1985).
- *Whole Earth Review* No. 67, Summer 1990.
- *Nick and the Glimmung* by Philip K. Dick (Piper Books, London, 1988).

Thanks to Jeff Barbose, Michael Greenspon, Bill Guschwan, Bo3b Johnson, Lisa Jongewaard, Ted Kaehler, and Ned van Alstyne (aka Ned Kelly) for reviewing this column.•

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

KON & BAL'S PUZZLE PAGE

FINDER++

See if you can solve this programming puzzle, presented in the form of a dialog between Konstantin Othmer (KON) and Bruce Leak (BAL). The dialog gives clues to help you. Keep guessing until you're done; your score is the number to the left of the clue that gave you the correct answer. These problems are supposed to be tough. If you don't get a high score, at least you'll learn interesting Macintosh trivia.

KONSTANTIN OTHMER AND BRUCE LEAK

KON I'm trying out this new C compiler to see what we can do to make the system and Finder smaller and faster.

BAL Wait. Since when has the Finder been written in C?

KON It's better than that! It's actually C++, with some assembly routines so that we can claim the copyright goes back to 1983.

BAL Oh, that explains how the System 7 Finder got so much bigger. I thought it was just the About box. I'm running System 6 on my PowerBook 100. I'd sure like to get a smaller and faster version of System 7. Are you making any progress?

KON Well, yes and no. The compiler output is certainly smaller but I haven't nailed down how much faster it is. When I boot up and the Finder launches, the machine restarts, which relaunches the Finder, which causes the machine to restart, and so on. It all happens pretty fast but doesn't seem all that useful.

BAL What machine is this on?

KON Macintosh Classic — the original, not the Macintosh Classic II.

BAL Somehow the compiler is generating bogus code that causes the system to restart. So I compare the code from the new compiler to the code from the old compiler, look at the differences, and see if they make sense.

KONSTANTIN OTHMER AND BRUCE LEAK Long-time Puzzle Page fan Al Gore recently invited KON and BAL to upgrade the White House situation room to BALKON-4, the latest in networked Spaceward Ho! technology. On his lunch hour, KON debugged Clinton's economic plan and found the memory leak that was causing that $50 billion Medicare shortfall. BAL is now working on an audio-animatronic Silicon Valley executive so that the President can always have one at his side.•

KON Everything's different: 42,000 bytes went away and the rest is totally different. This isn't a minor compiler revision. We're talking *Advanced Technology* here. Where are you going to look?

BAL OK, OK. Let's debug it. I set an ATB on _Launch and then another on _InitGraf.

KON OK. You break at _Launch and then after you Go you break at _InitGraf.

BAL I set an ATB on _WaitNextEvent.

KON You break at _WaitNextEvent.

BAL I say Go and see if I get back to _WaitNextEvent again.

100 KON The machine reboots almost immediately.

BAL I go back to the same place and instead of saying Go I trace over _WaitNextEvent.

95 KON The machine crashes into MicroBug. But it's not your ordinary crash into MicroBug. The screen is trashed and you can't type anything. But it looks as though MicroBug is trying to come up.

BAL Can I hit the NMI button?

90 KON You can press it all you want, but it doesn't do anything. And, by the way, G -1 doesn't work either.

BAL Hmmm. It seems as if something is seriously wrong with _WaitNextEvent. Did you recompile the Process Manager, any DAs, or other stuff?

85 KON Nope, I only recompiled the Finder. When I get that working, I'll get around to the rest.

BAL So you didn't recompile the Finder extensions? Since the C++ virtual function tables are different, all your existing Finder extensions are incompatible and maybe that's what's hosing you.

80 KON None of the extensions are active, and even if they were, the Finder verifies their versions. What do you expect? They're object oriented. Of course it works.

BAL Of course. Well, since we couldn't make it across _WaitNextEvent, let's step into it.

75 KON As soon as you step, you get the same weird crash into MicroBug.

BAL I just step into it?

KON Yes.

BAL As soon as I step, pending interrupts come in and kill me. So I disable interrupts with an SR = 27000000 and try stepping again.

70 KON Same crash.

BAL Seems like there might be something wrong with MacsBug.

KON Let me make sure I'm following you here. Only the Finder is recompiled and you blame the strange crashes on MacsBug? I'm going to have trouble selling that one.

BAL Clearly there's something wrong with the recompiled Finder. It's probably trashing MacsBug memory.

65 KON Come on. MacsBug does some sort of a checksum on itself and tells you if it's been altered. When you break at _WaitNextEvent, you don't get any messages to that effect.

BAL You got me there, KON. So you're saying that MacsBug is in perfect working order at this point. I can do an IL or whatever, but if I step I'm dead?

KON Perfect working order? Same as it ever was. But the Surgeon General has determined that stepping or tracing at this point causes ill effects.

BAL This is not my beautiful MacsBug. If I trace after I hit _InitGraf, is everything fine?

60 KON No problem.

BAL So I do an

```
ATB ';t ;g'
```

which breaks on every trap, traces over it, and then continues. That way I can see what the last trap I hit was.

55 KON The machine runs for a while, but when you crash and burn into MicroBug, you lose your MacsBug screen.

BAL Fine. I set up another screen, put MacsBug on that screen using the Monitors control panel, and use the SWAP command so that MacsBug is always visible. That way when I crash I can see what just happened.

KON Great strategy for a modular Macintosh, but this is on a Macintosh Classic. I'd let you figure it out that way except you used up your whole budget flying to North Dakota a few puzzles ago.

BAL I was hoping you'd forget that. OK, fine. Someone must be trashing low memory, so I'll use Bo3b Johnson's totally awesome Blat dcmd. It'll catch any read or write from memory locations $0–$100.

Bo3b Johnson's Blat dcmd can be found on this issue's CD and on the E.T.O. disc. (The "3" in Bo3b's name is silent.) Blat is written up in the Macintosh Debugging article in Issue 13 of *develop.*•

50 KON You're on a Macintosh Classic, which doesn't have an MMU. That dcmd works via the MMU.

BAL KON! Those correspondence classes are finally paying off. So I'll narrow down the area that's causing the problem by doing an ATB 10 to skip over 16 ($10) traps at a time until the machine crashes into MicroBug. If it takes five times to crash, the next time I'll do an ATB 40, and then an ATB 4, until it crashes. After I do this enough times I'll know what was the last trap that was successfully executed, and I can go from there.

45 KON Rather than crashing, the machine is now rebooting.

BAL OK, so what's the last trap called before the machine reboots?

40 KON _WaitNextEvent.

BAL Fabulous. Déjà vu. Is this a Never Ending Story? And when I'm at _WaitNextEvent I can't step or trace or anything?

KON Well, you can't step or trace. That's all you've tried so far.

BAL So I set a breakpoint on the first instruction of _WaitNextEvent and say Go.

35 KON You crash into MicroBug, just like before.

BAL OK, what's the current score? Can we call it quits?

KON I wouldn't say you aced this one. Luckily we're getting paid per word, so let's keep going.

BAL But when I was at _InitGraf, I could trace. So something's hosing MacsBug between _InitGraf and _WaitNextEvent. I'll do the ATB 10 trick like before, but this time I'll try tracing after every break. That way I can figure out where MacsBug is getting mauled.

30 KON You figure out that you can trace over a call to _InitWindows, but when you trace over the next trap, a call to _GetResource, you crash into MicroBug.

BAL So I go to _InitWindows and trace until I get to the call to _GetResource. If it's a long way, I do a T 1000. If that crashes, I reboot and do a T 500, then a T 250, and so on, until I find the offending instruction.

25 KON The offending instruction is a

```
MOVE.L d0,20(a2)
```

BAL What's in A2?

20 KON $100.

BAL Writing to low memory like this sounds like a bad idea. My guess is that A2 is trashed and we're pounding an important vector. What's at $120?

15 KON That's MacJmp.

BAL Aha! MacJmp is the vector that exception code uses to go to the debugger. Once you trash that, all bets are off.

KON Yeah, setting ATBs works because MacsBug patches the trap dispatcher and looks for the A-traps you have breaks on. If it encounters one, it just drops into MacsBug directly. Other breakpoints are set by replacing the existing instruction with a trap instruction. When these instructions are processed, they go through MacJmp. When MacJmp gets trashed, tracing and stepping and setting breakpoints no longer works, as we found out.

BAL Nasty.

KON Don't try to finish up so fast! You still haven't figured out why the machine is rebooting.

BAL The new compiler must do a better job of register allocation and actually use them all in its optimizations. Some Finder glue routine you called must have trashed A2.

KON Exactly. An easy problem to fix, though. The Finder was calling an assembly routine that hammered A2. After you fix the bug and build a new Finder, the machine still restarts.

BAL So I set an ATB on _WaitNextEvent, since that was as far as we got last time, and try to trace over it.

10 KON OK. No problem.

BAL Whew! Finally I get past that _WaitNextEvent. Let's go for two. I say Go and see if we hit _WaitNextEvent again.

KON Nope. The machine restarts.

BAL After the first _WaitNextEvent I do the trick with T 1000, T 500, T 250, and so on, until I find the offending instruction or subroutine. If the problem is occurring in a subroutine, I go into it and do the same thing. At some point this process has to stop and I'll find the problem instruction.

5 KON The offending instruction is an

```
LEA 13(a7),a7
```

SCORING

75–100	How long have you been a member of the Liar's Club?
50–70	Sharpshooter. You win the (virtual) kewpie doll.
25–45	A valiant effort. These puzzles are hard!
5–20	Brush up for Issue 15's Puzzle Page.•

BAL Well, that's bogus. Using an odd address on a 68000 will cause an address error.

KON Yeah, but the machine is rebooting.

BAL I get it. It's an odd address in the stack pointer. The Macintosh gets an address error because of the odd address. When it goes to process the exception, the exception handler gets an address error trying to push the exception frame onto the stack. If the Macintosh ran in user mode, it wouldn't have this problem, since it could switch to supervisor mode — essentially a clean machine with a properly aligned stack pointer — to handle the exception. But since it runs in supervisor mode, hosing the stack pointer messes the machine up to the point where it can't even handle an exception, so it reboots.

KON Yeah. We were working on cleaning up the stack after function calls in the compiler and had a small problem with the way Booleans are handled. Since a Boolean is only a char, which is one byte, the compiler thought it needed to clean up an odd amount of space from the stack. Once we explained to the compiler that stacks must be word aligned, the problem went away.

BAL Two bugs in one Puzzle Page!

KON Nasty.

BAL Yeah.

Thanks to Gary Davidian, scott douglass, and Jean-Charles Mourey for reviewing this column.•

INDEX

For a cumulative index to all issues of *develop*, see this issue's CD.•