develop

The Apple Technical Journal

Issue 12 December 1992

EDITORIAL STAFF

Editor-in-Cheek *Caroline Rose*

Technical Buckstopper *Dave Johnson*

Our Boss *Greg Joswiak*

His Boss *David Krathwohl*

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

Managing Editor *Monica Meffert*

Assistant Managing Editor *Ana Wilczynski*

Contributing Editors *Lorraine Anderson, 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, Lisa Jongewaard*

Online Production *Cassi Carpenter*

The stone jigsaw puzzle was pieced together by Hal Rucker of Rucker Huggins using Adobe Illustrator, FontMonger, Ray Dream Designer 2.0, and Adobe Photoshop.

develop, The Apple Technical Journal, is a quarterly publication of the Developer Support Information group.

The *Developer CD Series* disc for November/December 1992 or later contains this issue and all back issues of *develop* along with the code that the articles describe. The *develop* issues and code are also available on AppleLink and via anonymous ftp on ftp.apple.com.

CAROLINE ROSE

Dear Readers,

The more observant among you may have noticed that we've made yet another change to *develop* with respect to how it's dated. The last change happened with Issue 10, when we stopped designating issues with the current season and went back to using the current month, because the season isn't the same around the world. Now we've moved the date ahead by one month — also to accommodate worldwide distribution.

For the terminally curious, here are the details: *Apple Direct*, our vanguard of information for business and technical decison makers, doesn't reach other countries until two to eight weeks after it's distributed in the U.S.; it might, for example, be folded into a local mailing whose schedule doesn't coincide. So by the time some non-U.S. developers see *Apple Direct*, they think they've been sent a past issue rather than the latest one. To help convey to them that it is indeed the latest issue, it's now dated with a month that's closer to when they'll see it. The *Developer CD Series* disc, *Apple Direct*, and *develop* all need to be in sync — so there you have it. What is now the December issue of *develop* was the Autumn issue last year and the October issue in 1990 (when our production cycle was a month out of phase from where it is now). Anyway, we hope those of you in the U.S. agree there's no harm in a little time travel forward.

A little time travel forward would be really handy for me while I'm writing these editorials, because I don't always know what the state of the *develop*-related world will be two months in advance (that's the lag time before you actually read this). In Issue 11's editorial, for example, I couldn't alert you to *develop*'s being in a new format on the *Developer CD Series* disc, because at that time we weren't sure it would make it onto that disc. Yes, we've responded to your complaints about *develop* in HyperCard® by switching to that popular viewing tool that you may know as "BlueNote" — now "Apple DocViewer" — the same tool that's used for viewing *New Inside Macintosh*.

The Developer CD corresponding to Issue 11 contained a prerelease version of DocViewer that still needed some work; for example, it wouldn't work at all on a Macintosh Plus. In lieu of a time machine, I've consulted the Magic 8-Ball DTS uses to answer developer questions, and it tells me that the CD corresponding to this issue of *develop* — called the "November/December" CD, to ease the transition — will include a version 1.0 release of DocViewer along with Issues 11 and 12 in DocViewer

CAROLINE ROSE (AppleLink CROSE) has been writing software documentation since before there were personal computers or even lava lamps. Her total of five years at Apple is (to use the jargon she helped coin in *Inside Macintosh* Volume I) a discontinuous selection, interrupted by as many years at NeXT. When not reading, writing, coining, or otherwise obsessing over words, Caroline enjoys the outdoors. (As songwriter Greg Brown puts it, "People say small things when they stay too long in little rooms.") The highlight of her summer was "swimming Lava Falls": being thrown from a raft that capsized in the largest rapid (a 37-foot drop) on the Colorado River in the Grand Canyon, and being rescued by a small paddleboat that braved the next rapid with 12 worried souls aboard. Talk about an adrenaline rush! And she lived to tell the tale.•

format. Version 1.0 should work on Macintosh Plus and newer models, with system software version 6.0 and later. Back issues of *develop* will eventually also make their way over into this format (the 8-Ball is hazy regarding just when this will happen). We'd really like your feedback on DocViewer and how well it works for reading *develop* (or anything else). Please check it out, and send your flames or even praise to AppleLink DEV.CD.

Whoops — did I say "DTS"? Old habits die hard. Another change we're gradually making in *develop* is to shift from "Developer Technical Support" (DTS) to "Developer Support Center" (DSC). As you may have read in the April 1992 issue of *Apple Direct*, the DSC is a gateway to DTS as well as other support-related resources. It provides a focal point for developer queries — a single AppleLink address, DEVSUPPORT, and a single phone number, (408)974-4897. Developers who aren't Apple Associates or Partners can contact the DSC for limited nontechnical support and referrals. We'll be adjusting to this change along with others that are creeping in: Tech Note references no longer numbered; *Inside Macintosh* references that include *New Inside Macintosh*; DocViewer as the on-line viewing tool; postdating; and other changes that I foresee but don't dare reveal lest I upset the delicate balance of the universe.

Finally, I feel compelled to explain my bizarre trivia answer in Issue 11, about the upside-down character that wasn't. I claimed the offending character was "8," which on the contrary looks perfectly OK — not at all topheavy — in printed *develop*. It turns out that this "8" is topheavy only in LaserWriter output. That will teach me to use a media-specific question! I think I'll quit while I'm behind and lay off trivia questions altogether for a while (even though I'll miss those friendly letters from you).

CRose

Caroline Rose
Editor

SUBSCRIPTION INFORMATION

To subscribe to *develop*, 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 *Developer CD Series* disc.•

LETTERS

POSTAL DEVILS EATING CDS?

develop is the most exciting piece of regular mail I get after Japanese animation laserdiscs. I joyfully received Issue 11 but unfortunately the wolverines in the Postal Service dined on some of the plastic and no CD was to be found! Help!

— Jim Perry

Would you consider mailing *develop* in a nonperforated plastic wrapping? The perforation was two-thirds torn when I received it.

— Eva Schlesinger

I really enjoy *develop*, but I have to say that I've enjoyed it less recently.

Some time ago the CDs came in a small envelope well protected inside the magazine, and everything was fine. Now, *develop* is shipped with the CD in its own holder, which would seem to be a fabulous idea except that you were blind-sided by the U.S. Post Office.

Every month since the CD got its own holder, the Post Office has mangled my plastic bag, CD holder, and magazine. Today my *develop* issue 11 arrived sans CD. I called the subscription office (1-800-545-9364) and they promise to send me another within four weeks.(!?) Growl.

— Bob Cent

Most of the mail I get is, unfortunately, on this subject. Our Production Manager, Hartley Lesser, really has been working on it. Even with Issue 11, we took a small step toward solving the problem: since many people thought someone was breaking open the package and stealing the CD, we inserted a thick sheet of paper over the CD so that it wouldn't be visible. But complaints of torn packaging still came in, so clearly the packaging just wasn't sturdy enough. The packaging around Issue 12 and its CD should be about twice as thick as before and have no perforation. If that doesn't work we'll try something else.

Issue 7 was the last one to list the 800 number you used to contact the subscription office (though it stubbornly has still shown up on our renewal notice). The correct number is 1-800-877-5548. The person you spoke to normally doesn't handle calls regarding develop *and didn't know that replacement CDs should be mailed within a day or two of notification of the problem. Sorry for the mixup. We hope you'll never need that service again!*

— Caroline Rose

SCREENWRITING CAVEAT

Your Issue 11 column on drawing to the screen was really useful to me. I had an animation program that wrote directly to the screen and it worked fine. But when I upgraded to a new accelerator card my program kept crashing. I spent months trying to figure out the problem. But your article fixed it straight away. All I needed was the SwapMMUMode calls. I don't know why the previous card didn't require them, but my program works fine now.

— Tony Cooper

Thanks for your interest in the column. We're glad it was helpful to you.

One thing we want to be sure to mention is that writing directly to the screen will break for sure on future Macintosh systems based on RISC technology. And we again want to

WHY DON'T YOU WRITE MORE OFTEN?
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).•

stress that the only applications that should even consider writing directly to the screen are games and other animation programs.

— Brigham Stevens and Bill Guschwan

USER-FRIENDLY RENEWING

Recently I received a couple of renewal notices for *develop* in the mail. In trying to decipher these notices, I realized that user friendliness is something we should all be aiming for not just in the software we write, but in everything we do. It's interesting how working with the Macintosh makes one aware of human interface issues in everyday life.

Anyway, I think there are a few ways in which the *develop* renewal notices could be made more user friendly:

1. Leave a bigger space for writing the credit card number.

2. Clearly indicate on the renewal notice the date my subscription expires.

3. Is there any reason why the renewal notices are printed in red ink?

—Tim Hammett

We're in the process of making the changes you suggested to the develop *renewal notice.*

1. We'll leave a bigger space for writing the credit card number.

2. The notice will indicate when the subscription expires. You can also find this out at any time from your mailing label: the number that appears on a line by itself at the top of the label indicates the last issue you'll receive unless you renew.

3. The reason for the red ink is so that this little piece of paper doesn't get lost on your desk. But you've inspired us to change it to a more readable, deeper red.

We're also correcting the 800 phone number on the notice, to 1-800-877-5548.

Thanks for your letter. Without it, I would have assumed that the renewal notice (which isn't really in my domain) was in great shape. I appreciate the enlightenment.

—Caroline Rose

REUSED CDS: IS IT ART?

In Issue 10 of *develop*, Bruce Radford stated that he wasn't sure what to do with his old CDs. He felt that he should recycle them, but he wasn't sure how. Well, I have a suggestion.

Many people forget that reusing something is often even better than straight recycling. My school would have many uses for old issues of the *develop* CD. I know a few friends who would love copies, no matter how old; I could use them in a programming class; and other students could cut them up to make jewelry for school fundraisers. I also have many uses for old 256K SIMMs, which seem to be becoming about as useful as pennies now.

So go ahead and send the stuff that you think no one needs to me, or to a school near you.

— Peter Bierman (age 16)
BS Software
5757 Olentangy Blvd.
Worthington, OH 43085

Thanks for the idea. Day care centers and children's museums have also been mentioned as possible destinations for old CDs. We suggest that before giving away CDs for for art projects, developers put a deep scratch through the data side of the CD if it contains any confidential or licensed data.

For some wild and crazy ideas on this from Apple's Developer Support Center, see the Q & A on page 126.

— Caroline Rose

DEVELOP INTERNET ADDRESS

I'm on the Internet and *develop* contains only AppleLink addresses. I'm guessing that crose@applelink.apple.com is your Internet address. *develop* really should have an Internet address for academic developers to send e-mail to.

— Eric Kofoid

Adding "applelink.apple.com" to any AppleLink address converts it to an Internet address. The Internet addresses for me and develop*'s Technical Editor Dave Johnson are listed on the last page of every issue.*

— Caroline Rose

BACK ISSUES CONUNDRUM

I noticed that your back issues are listed at $13 in *develop* and at $10 in the APDA catalog.

Why the discrepancy? Who should I order the back issues from?

— Michael Tackie

P.S. Great magazine. Very technical. I don't understand everything, but that's good; it forces me to become a better programmer.

You pay a $3 shipping charge when you order from APDA, so it adds up to $13 in the end.

— Caroline Rose

P.S. Thanks!

CORRECTION TO APPLE EVENTS ARTICLE IN ISSUE 10

The "Apple Event Objects and You" article in *develop* Issue 10 contains two errors in the printed sample code. The first problem is that five lines were omitted from the end of GetWindowIndex. The code at the top of page 25 should be changed from

```
      return noErr;
}
```

to

```
if ((rawIndex > numWindows)||(rawIndex <= 0)) {
   *index = 0;
   return errAENoSuchObject;
} else
   *index = rawIndex;
   return noErr
}
```

The second bug is in the routine WriteRectToken (page 30). The following call

```
BlockMove(*thisRectDesc.dataHandle,
   &tokenPtr->theRect,sizeof(Rect));
```

should be changed to

```
BlockMove(*thisRectDesc.dataHandle,
   (Ptr)tokenPtr->theRect,sizeof(Rect));
```

Since theRect is actually a pointer to a rectangle (see the declaration at the top of page 29), the first version would have destroyed the pointer and four bytes of the following long integer.

Thanks to Doug McKenna, the author of Resorcerer, for pointing out these problems.

TECHNIQUES FOR WRITING AND DEBUGGING COMPONENTS

Programmers first saw the Component Manager as part of the QuickTime 1.0 system extension. Now that the Component Manager is part of System 7.1, components aren't just for QuickTime programmers any more. This article shows you how to take advantage of the power and flexibility of components as a way to give extended functionality to any Macintosh application.

GARY WOODCOCK AND CASEY KING

Software developers are continually searching for ways to avoid reinventing the proverbial wheel every time they need new capabilities for their programs. A new approach is available with components. Components are modules of functionality that applications can share at run time. They enable applications to extend the services of the core Macintosh system software with minimal risk of introducing incompatibilities (unlike, for example, trap patching).

As Figure 1 suggests, components also encourage a building-block approach to solving complex problems. Higher-level components can call lower-level components to build sophisticated functionality, while at the same time making the application program interface (API) much simpler. What's more, because components are separate from an application that uses them, you can modify and extend components without affecting the application.

Components are maintained by the Component Manager, which is responsible for keeping track of the components available at any given time and of the particular services they provide. The Component Manager provides a standard interface through which applications establish connections to the components they need.

Almost anything you can dream up can be a component — video digitizer drivers, dialogs, graphics primitives, statistical functions, and more. QuickTime 1.0 itself contains a number of useful components, including the movie controller, the sequence grabber, and a variety of image compressors and decompressors (*codecs*), all of which are available to any client application.

GARY WOODCOCK AND CASEY KING have a long history of collaboration. They first met at a flight simulation company in the early 80's where they worked together on designing a multimillion-dollar F-16 jet fighter simulator (and you thought Falcon was cool!). They parted ways temporarily, but regrouped at Apple to join forces in what colleague Jim Batson has termed the "QuickTime sleep deprivation experiment." They're both currently working on RISCy products, but from different parts of the country (Gary in Cupertino, and Casey in the new PowerPC mecca of Austin, Texas). With his wife Lonna, Casey is the proud co-owner of his latest obsession — a year-old baby boy named Brian — but he still makes time for mountain biking, hiking, and flying. Gary still spends much of his time diligently testing video capture cards for

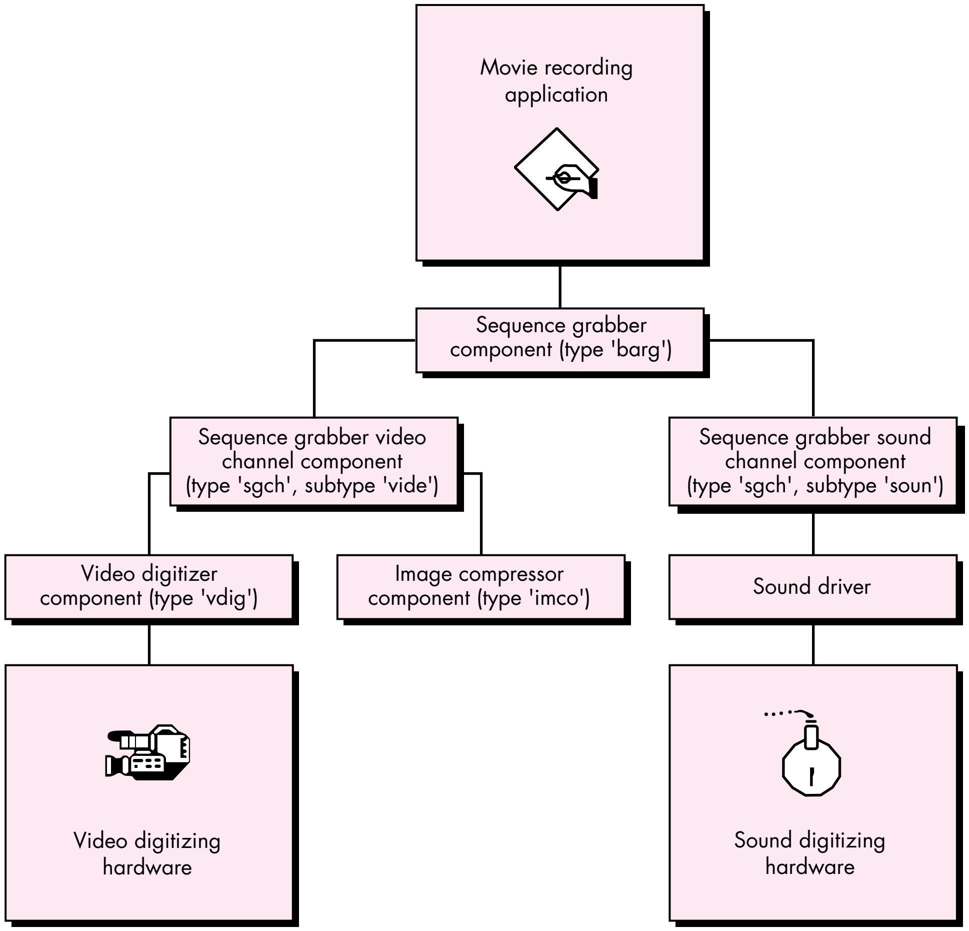

Movie recording application
Sequence grabber component (type 'barg')
Sequence grabber video channel component (type 'sgch', subtype 'vide')
Sequence grabber sound channel component (type 'sgch', subtype 'soun')
Video digitizer component (type 'vdig')
Image compressor component (type 'imco')
Sound driver
Video digitizing hardware
Sound digitizing hardware

Figure 1
Using Components as Software Building Blocks

To demonstrate the all-around usefulness of components, we'll examine the development and implementation of a component that does some rather trivial mathematical calculations. This example will help us focus on concepts rather than getting lost in the details of solving a complex problem. We'll build a fairly generic component template that you can use in your own designs. We'll also discuss some advanced component features, such as extending component functionality, capturing components, and delegating component functions. Finally, we'll show you some techniques and tools for debugging your components. The accompanying *Developer CD Series* disc contains our example component's source code, a simple application to test our component, and the debugging tools.

QuickTime compatibility with Movie Recorder (translation: watching *Star Trek: The Next Generation* episodes on his Macintosh). Occasionally he ventures out for a bit of mountain biking or flying. This article is their latest joint venture.•

COMPONENT TRIVIA #1

The original name for the Component Manager (as conceived of by Bruce "Of *course* the feature set is frozen!" Leak) was the Thing Manager. Components were referred to as "things" (as were the QuickTime project schedules, the significance of which engineers couldn't easily grasp). The use of this terminology led to one of two conditions in most QuickTime engineers: in some, an irrepressible compulsion to make "thing" puns, and in others, perhaps as a backlash against the former, an almost pathological aversion to the use of the word "thing" in normal conversation.

Note that this article doesn't spend a great deal of time explaining how applications can find and use components. We assume that you've invested some effort in reading the *QuickTime Developer's Guide* (part of the QuickTime Developer's Kit). If you haven't, we strongly urge you to do so, since the *Developer's Guide* contains the definitive description of the Component Manager.

SHOULD YOU WRITE A COMPONENT?

OK, components sound interesting, but should you write one? Why write a component when you can just code the functionality you need directly into your application or write a device driver? Here are a few reasons to choose components over the alternatives:

- Components are easier for applications to use. Client applications don't have to know what they're looking for before opening a service. This is different from device drivers, where open calls must provide either a driver name or a refNum. An application can simply tell the Component Manager, "I'm looking for somebody to do this for me. Is anybody available?" In addition, clients don't need to set up parameter blocks or make control/status calls to use components. Armed with the API of the component type, the caller simply makes normal function calls to the component, and the Component Manager does the work.
- Components are more flexible. You can modify the behavior of a component by overriding its capabilities without adversely affecting the application. The Component Manager enables the component to communicate its capabilities to clients dynamically.
- Components allow you to design more flexible applications. They can be used to divide the functional aspects of an application into parts. For example, a word processing application might use a spelling checker component, a thesaurus component, and a grammar checker component. If the thesaurus component is

updated, the application code doesn't have to change at all. A user can simply replace the old thesaurus component with the new one.

- Components are easier to implement than device drivers. There are no declaration structures, driver headers, assembly code glue, installation INITs, or any of the peculiarities that come with device drivers.
- Components are easier to debug than device drivers. No longer will you be walking the unit table to find your driver so that you can set a breakpoint at your control call dispatcher. You can easily and effectively debug your code using a source-level debugger such as Symantec's THINK C Debugger.

Now that you know the advantages of components, you have to decide whether the functionality you need is a good candidate for a component. To do this, ask yourself the following:

- Do I anticipate reusing this functionality in other applications? Components are ideal for providing services that many applications can use.
- Do I anticipate having to modify certain aspects of this functionality in the future? Functionality encapsulated in a component can be extended or modified without disturbing the original interface specification.
- Is there a benefit to users in establishing a common API for this functionality, so that other developers can use or extend it? You might want to be able to allow third parties to extend your application without having to expose detailed information about your application's internal data structures. For example, many of the "plug-in" modules for today's popular graphics applications could easily be implemented as components.

A "yes" to more than one of these questions means that components are probably a good approach for your next product. But you still have one last question to answer: has someone else already written a component that solves your problem? To find out, you need to contact Apple's Component Registry group (AppleLink REGISTRY) and ask them. These folks maintain a database of all registered component types, subtypes, and manufacturers, as well as the corresponding APIs (if they're publicly available). A check with the Registry is mandatory for anyone who's contemplating writing a component.

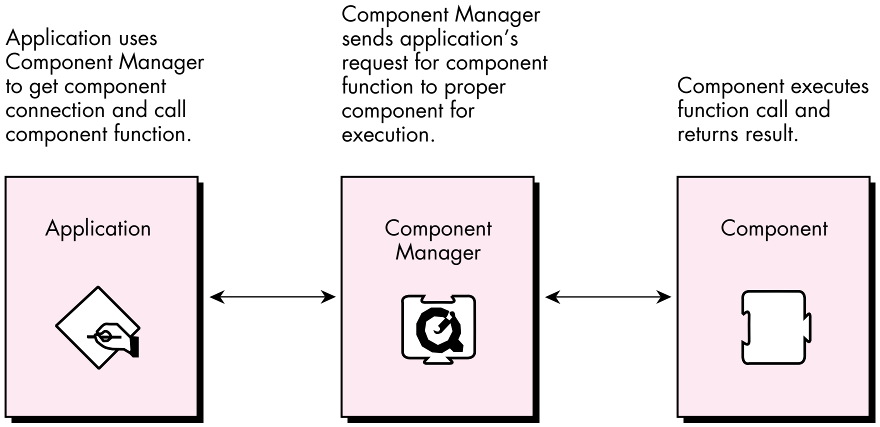
If after all this you find that you're still about to embark into uncharted territory, read on, and we'll endeavor to illuminate your passage.

COMPONENT TRIVIA #2

The original component type for the sequence grabber component was, logically enough, 'grab'. The engineer primarily responsible for the sequence grabber, Peter Hoddie, requires massive infusions of Diet Coke to function properly. During a particularly intense bout of engineering mania, the Diet Coke supply was exhausted; unbeknownst to anyone, Peter became temporarily dyslexic and changed the sequence grabber component type to 'barg'. The change was never noticed, and it caused no real harm, other than the wasted time developers spent trying to figure out what 'barg' might be an acronym for (Boffo Audio Reverb Gadget? Bodacious Analog Reference Gizmo?). Peter's brain has since returned to its (relatively) normal state.

COMPONENT BASICS 101

Client applications use the Component Manager to access components. As shown in Figure 2, the Component Manager acts as a mediator between an application's requests for component services and a component's execution of those requests. The Component Manager uses a *component instance* to determine which component is needed to satisfy an application's request for services. An instance can be thought of as an application's connection to a component. We'll have more to say about component instances later on.



Application uses Component Manager to get component connection and call component function.

Component Manager sends application's request for component function to proper component for execution.

Component executes function call and returns result.

Figure 2
How Applications Work With Components

Conceptually, components consist of two parts: a collection of functions as defined in the component's API, and a dispatcher that takes care of routing application requests to the proper function. These requests are represented by request codes that the

Component Manager maps to the component functions. Let's take a look at both the component functions and the component dispatcher in detail.

COMPONENT FUNCTIONS

There are two groups of functions that are implemented in a component. One group does the custom work that's unique to the component. The nature of these functions depends on the capabilities that the component is intended to provide to clients. For example, the movie controller component, which plays QuickTime movies, has a number of functions in this category that control the position, playback rate, size, and other movie characteristics. Each function defined in your component API must have a corresponding request code, and you must assign these request codes positive values (0 or greater).

The second group of functions comprises the standard calls defined by the Component Manager for use by a component. Currently, four of these standard calls *must* be implemented by every component: open, close, can do, and version. Two more request codes, register and target, are defined, but supporting these is optional. The standard calls are represented by negative request codes and are defined*only* by Apple.

Here's a quick look at each of the six standard calls.

The open function. The open function gives a component the opportunity to initialize itself before handling client requests, and in particular to allocate any private storage it may need. Private storage is useful if your component has hardware-dependent settings, local environment settings, cached data structures, IDs of component instances that may provide services to your component, or anything else you might want to keep around.

The close function. The close function provides for an orderly shutdown of a component. For simple components, closing mainly involves disposing of the private storage created in the open function. For more complex components, it may be necessary to close supporting components and to reset hardware.

The can do function. The can do function tells an application which functions in the component's API are supported. Clients that need to query a component about its capabilities can use the ComponentFunctionImplemented routine to send the component a can do request.

The version function. The version function provides two important pieces of information: the component specification level and the implementation level. A change in the specification level normally indicates a change in the basic API for a particular component class, while implementation-level changes indicate, for example, a bug fix or the use of a new algorithm.

The register function. The register function allows a component to determine whether it can function properly with the current system configuration. Video digitizer components, for example, typically use register requests to check for the presence of their corresponding digitizing hardware before accepting registration with the Component Manager. A component receives a register request code only if it explicitly asks for it. We'll see how this is done when we walk through our sample component.

The target function. The target function informs your component it has been *captured* by another component. Capturing a component is similar to subclassing an object, in that the captured component is superseded by the capturing component. The captured component is replaced by the capturing component in the component registration list and is no longer available to clients. We'll discuss the notion of capturing components in more detail later.

THE COMPONENT DISPATCHER

All components must have a main entry point consisting of a dispatcher that routes the requests the client application sends via the Component Manager. When an application calls a component function, the Component Manager passes two parameters to the component dispatcher — a ComponentParameters structure and a handle to any private storage that was set up in the component's open function. The ComponentParameters structure looks like this:

```
typedef struct {
   unsigned char  flags;
   unsigned char  paramSize;
   short          what;
   long           params[kSmallestArray];
} ComponentParameters;
```

The first two fields are used internally by the Component Manager and aren't of much interest here. The what field contains the request code corresponding to the component function call made by the application. The params field contains the parameters that accompany the call.

Figure 3 shows a detailed view of how a component function call from an application is processed. The component dispatcher examines the what field of the ComponentParameters record to determine the request code, and then transfers control to the appropriate component function.

REGISTERING A COMPONENT

Before a component can be used by an application, it must be registered with the Component Manager. This way the Component Manager knows which components are available when it's asked to open a particular type of component.

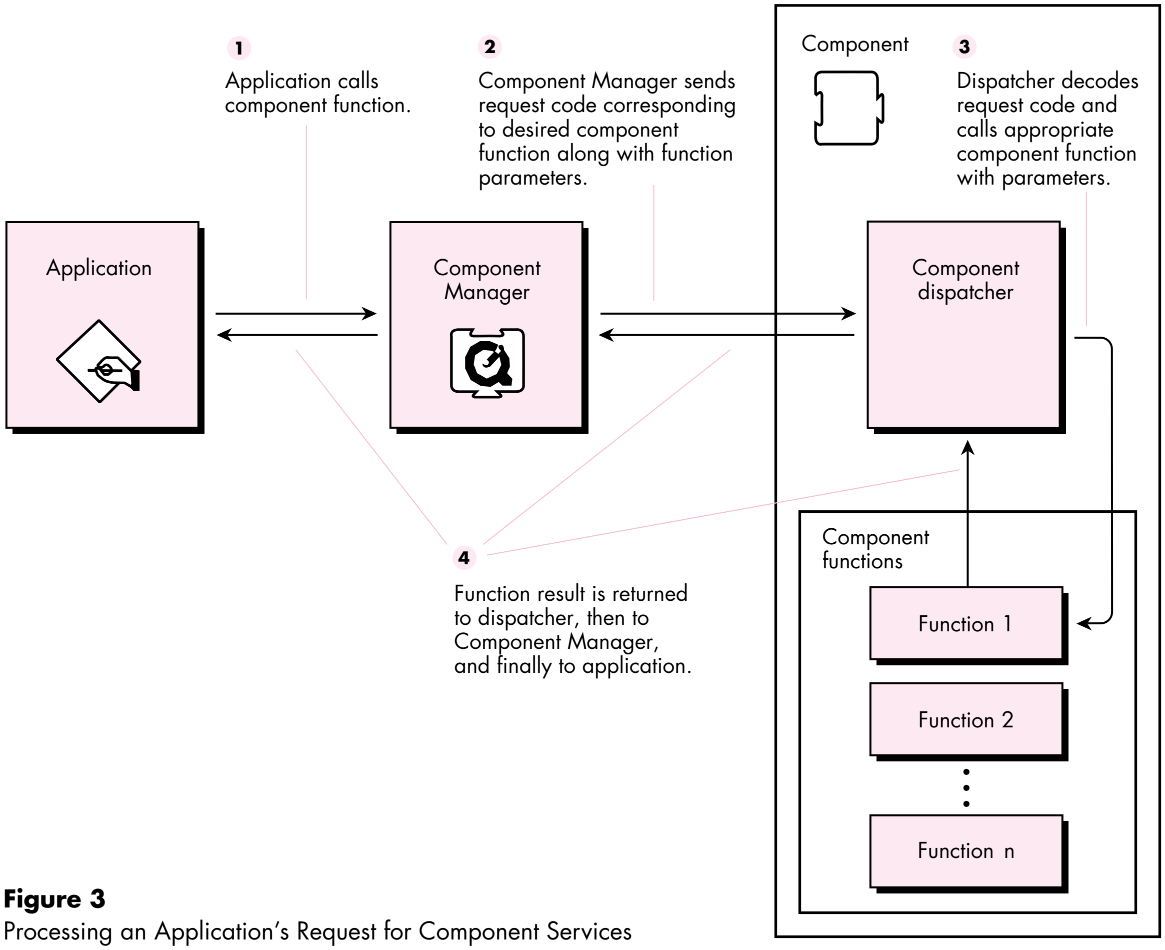

1
Application calls component function.
2
Component Manager sends request code corresponding to desired component function along with function parameters.
Component
3
Dispatcher decodes request code and calls appropriate component function with parameters.
Application
Component Manager
Component dispatcher
Component functions
Function 1
Function 2
Function n
4
Function result is returned to dispatcher, then to Component Manager, and finally to application.

Figure 3
Processing an Application's Request for Component Services

Autoregistration versus application registration. There are two ways that you can register a component. By far the easiest way is to build a standalone component file of type 'thng'. At system startup, the Component Manager will automatically register any component that it finds in files of type 'thng' in the System Folder and in the Extensions folder (in System 7) and its subfolders. The 'thng' component file must contain both your component and the corresponding component ('thng') resource. The definition of this resource can be found in the Components.h header file and is shown below.

```
typedef struct {
   unsigned long  type;        /* 4-byte code */
   short          id;
} ResourceSpec;
```

```
typedef struct {
   ComponentDescription td;            /* Registration parameters */
   ResourceSpec         component;     /* Resource where code is found */
   ResourceSpec         componentName; /* Name string resource */
   ResourceSpec         componentInfo; /* Info string resource */
   ResourceSpec         componentIcon; /* Icon resource */
} ComponentResource;
```

Figure 4 shows the contents of the component resource that we'll use for the example component.

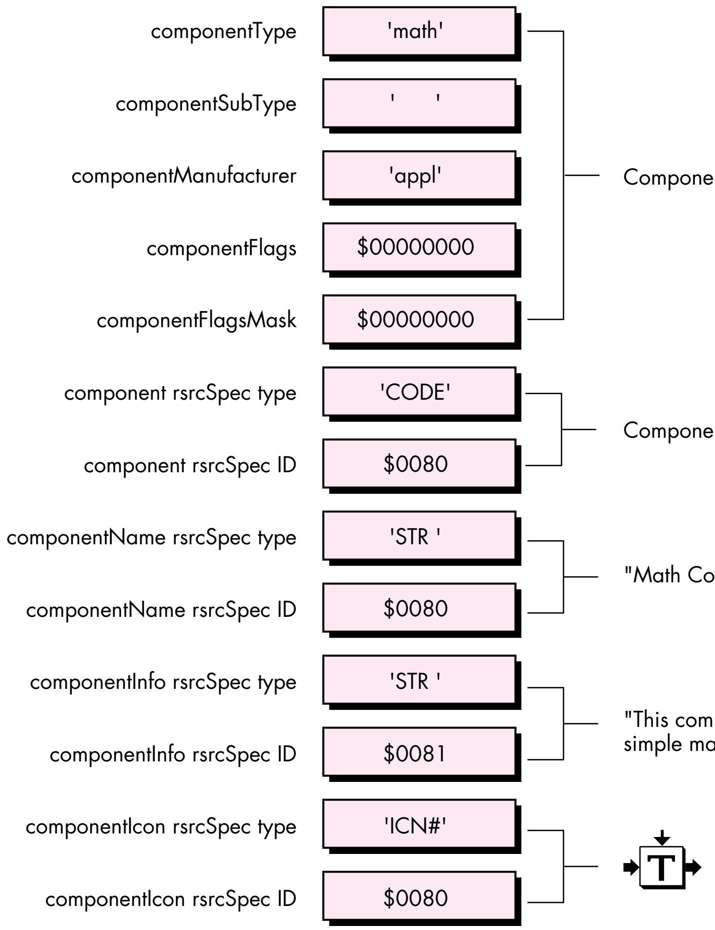


Figure 4
Math Component Resource

An application can also register a component itself using the Component Manager call RegisterComponent or RegisterComponentResource. As we'll see, this registration method facilitates symbolic debugging of components.

Global versus local registration. Components can be registered locally or globally. A component that's registered locally is visible only within the A5 world in which it's registered, whereas a globally registered component is available to all potential client applications. Typically, you register a component locally only if you want to restrict its use to a particular application.

A SIMPLE MATH COMPONENT

To help you understand how to write a component, we're going to go through the whole process with an example — in this case, a simple math component. We start by contacting the Apple Component Registry group, and to our astonishment (and their bemusement), we find that there are no registered components that do simple math! We assume for the moment that the arithmetic operators in our high-level programming language are unavailable and that our application is in desperate need of integer division and multiplication support.

We create a component called Math that performs integer division and multiplication.

THE FUNCTION PROTOTYPE DEFINITION

We need to define function prototypes for each of the calls in our component API — namely, DoDivide and DoMultiply. The function prototype for the DoDivide component call can be found in MathComponent.h and is shown below. The declaration for the DoMultiply function is similar.

```
pascal ComponentResult DoDivide (MathComponent mathInstance,
   short numerator, short denominator, short *result) =
   ComponentCallNow (kDoDivideSelect, 0x08);
```

This resembles a normal C language function prototype with a relatively straightforward parameter list. The mathInstance parameter is the component instance through which the application accesses the component; we'll see how an application gets one of these instances in a moment. The numerator and denominator parameters are self-explanatory and are passed in by the calling application as well. The contents of the last parameter, result, are filled in by the DoDivide function upon completion.

Those of you who have a passing familiarity with C are probably more than a little curious about the last portion of the declaration. ComponentCallNow is a macro defined by the Component Manager (see "Inside the ComponentCallNow Macro" for the nuts and bolts of how the macro works). Its main purpose is to identify a

INSIDE THE COMPONENTCALLNOW MACRO

Some of you may be wondering exactly what the ComponentCallNow macro does. Let's expand this macro for our DoDivide component call and examine it in detail.

```
= {0x2F3C, 0x08, kDoDivideSelect, 0x7000,
   0xA82A};
```

The first element, 0x2F3C, is the Motorola 68000 opcode for a move instruction. Execution of this instruction loads the contents of the next two elements onto the stack. The next element, 0x08, is the amount of stack space that we calculated for the function parameters of the DoDivide call. The third element, kDoDivideSelect, is the request code corresponding to the DoDivide call. The fourth element, 0x7000, is the Motorola 68000 opcode for an instruction that sets the contents of register D0 to 0. The Component Manager interprets this condition as a request to call your component rather than handling the request itself. The last element, 0xA82A, is the opcode for an instruction that executes a trap to the Component Manager.

While you can use this inline code in your component function declarations directly, we recommend that you use the ComponentCallNow macro to make your code more portable.

routine as a component function, as opposed to a normal C function. When an application calls the DoDivide function, the macro is executed. This causes a trap to the Component Manager to be executed, allowing the Component Manager to send a message to the component responsible for handling the function.

The first parameter to the ComponentCallNow macro is an integer value representing the request code for the integer division function. As noted earlier, your component's dispatcher uses this request code to determine what function has been requested. Recall that you may only define request codes that are positive.

The second parameter is an integer value that indicates the amount of stack space (in bytes) that's required by the function for its parameters, not including the component instance parameter. Be careful to note that Boolean and single-byte parameters may need to be passed as 16-bit integer values (see the section "Eleven Common Mistakes" for details). For the Math component, the space required for the DoDivide function is two 16-bit integers followed by a 32-bit pointer, for a total of eight bytes.

THE MATH COMPONENT DISPATCHER

The dispatcher of the Math component is shown in its entirety below. Notice that the dispatcher executes its component functions indirectly by calling one of two Component Manager utility functions — CallComponentFunction or CallComponentFunctionWithStorage. You use CallComponentFunction when your component function needs only the fields in the ComponentParameters structure, and CallComponentFunctionWithStorage when it also needs access to the private storage that was allocated in your component's open function.

```
pascal ComponentResult main (ComponentParameters *params,
                                 Handle storage)
{
   // This routine is the main dispatcher for the Math component.
   ComponentResult result = noErr;

   // Did we get a Component Manager request code (< 0)?
   if (params->what < 0) {
      switch (params->what)
      {
         case kComponentOpenSelect:       // Open request
            result = CallComponentFunctionWithStorage (storage, params,
                        (ComponentFunction) _MathOpen);
            break;
         case kComponentCloseSelect:      // Close request
            result = CallComponentFunctionWithStorage (storage, params,
                        (ComponentFunction) _MathClose);
            break;
         case kComponentCanDoSelect:      // Can do request
            result = CallComponentFunction (params,
                        ComponentFunction) _MathCanDo);
            break;
         case kComponentVersionSelect:    // Version request
            result = CallComponentFunction (params,
                        (ComponentFunction) _MathVersion);
            break;
         case kComponentTargetSelect:     // Target request
            result = CallComponentFunctionWithStorage (storage, params,
                        (ComponentFunction) _MathTarget);
            break;
         case kComponentRegisterSelect:   // Register request not
                                          // supported
         default:                         // Unknown request
            result = paramErr;
            break;
      }
   }
   else {                                 // One of our request codes?
      switch (params->what)
      {
         case kDoDivideSelect:            // Divide request
            result = CallComponentFunction (params,
                        (ComponentFunction) _MathDoDivide);
            break;
         case kDoMultiplySelect:          // Multiply request
            result = CallComponentFunction (params,
                        (ComponentFunction) _MathDoMultiply);
            break;
         default:                         // Unknown request
            result = paramErr;
            break;
      }
   }
   return (result);
}
```

A drawback of the dispatcher is the overhead incurred in having the Component Manager functions mediate all your requests. To reduce your calling overhead and thus improve performance, you can use a *fast dispatch* technique. While this technique is used in most of the QuickTime 1.0 components, this is the first time that it's been publicly described. See "Fast Component Dispatch" for details.

THE MATH COMPONENT DODIVIDE CALL

For the Math component, the DoDivide function is declared as follows:

```
pascal ComponentResult _MathDoDivide (short numerator, short denominator,
                                       short* quotient)
{
   ComponentResult result = noErr;

   if (denominator != 0) {
      *quotient = numerator/denominator;
   }
   else {
      *quotient = 0;
      result = -1L;  // Divide by zero not allowed
   }
   return (result);
}
```

The key thing to note here is that component functions must always return a result code. The return value is 32 bits and is defined in the API for the component. In our case, a value of 0 (noErr) indicates successful completion of the call and a negative value indicates that an abnormal completion occurred. Note that for some components a negative result code could indicate that the returned parameter values should be interpreted in a particular manner. For example, a video digitizer may return a negative result code of notExactSize from the VDSetDestination call. This doesn't indicate an error. It just means that the requested size wasn't available on the digitizer and that the next closest size was given instead. Also, since this result code is

FAST COMPONENT DISPATCH

BY MARK KRUEGER

If you're concerned about the time it takes to dispatch calls made to your component, try the fast dispatch method. This method eliminates the need for your component to make the extra call to the Component Manager functions CallComponentFunction and CallComponentFunctionWithStorage, and allows control to pass directly back to the caller. It does this by calling your component entry point with the call's parameters, the instance storage, and the caller's return address already on the stack. It passes the component request code in register D0, and points register A0 at the stack location where the instance storage is kept.

To handle a fast dispatch, you must write your component entry point in assembly language. Use the request code in D0 as an index into a table of function addresses, paying special attention to the negative request codes used for the standard Component Manager calls like OpenComponent and CloseComponent. If the functions are defined correctly, the dispatcher can jump directly to the function address. Note that the function parameter the caller uses to specify the component instance will instead be a handle to your component instance storage. When the function completes, control will return to the calling application.

You need to tell the Component Manager that your component has a fast dispatch handler instead of a normal dispatcher. To do this, set bit 30 ($40000000) of the componentFlags field of your component resource, and the Component Manager will always call your component using the fast dispatch method.

32 bits, you could actually return pointers or handles as results, rather than error codes.

USING THE MATH COMPONENT

In this section, we look at how an application uses the Math component. First, the application has to ask the Component Manager to locate the Math component. If the Math component is found, the application can open it and make calls to it.

FINDING AND OPENING THE MATH COMPONENT

We tell the Component Manager which component we're looking for by sending it a ComponentDescription record containing the type, subtype, and manufacturer codes for the desired component. We then call the Component Manager routine FindNextComponent to locate a registered component that fits the description. The code fragment below shows how this looks.

```
ComponentDescription    mathDesc;
Component               mathComponentID;

// Math component description
mathDesc.componentType = mathComponentType;
mathDesc.componentSubType = 0L;              // Wild card
mathDesc.componentManufacturer = 'appl';
mathDesc.componentFlags = 0L;              // Wild card
mathDesc.componentFlagsMask = 0L;          // Wild card

// Find a Math component
mathComponentID = FindNextComponent (nil, &mathDesc);
```

The zeros in the componentSubType, componentFlags, and componentFlagsMask fields indicate that they function as wild cards. If the Component Manager was unable to locate a component matching the description, it returns zero.

Assuming the Component Manager returned a nonzero component ID, we now open the component using the OpenComponent call, as follows:

```
mathInstance = OpenComponent (mathComponentID);
```

OpenComponent returns a unique connection reference — a component instance — to the Math component. If the component instance is nonzero, we're ready to use the component. Figure 5 illustrates the process of finding a component.

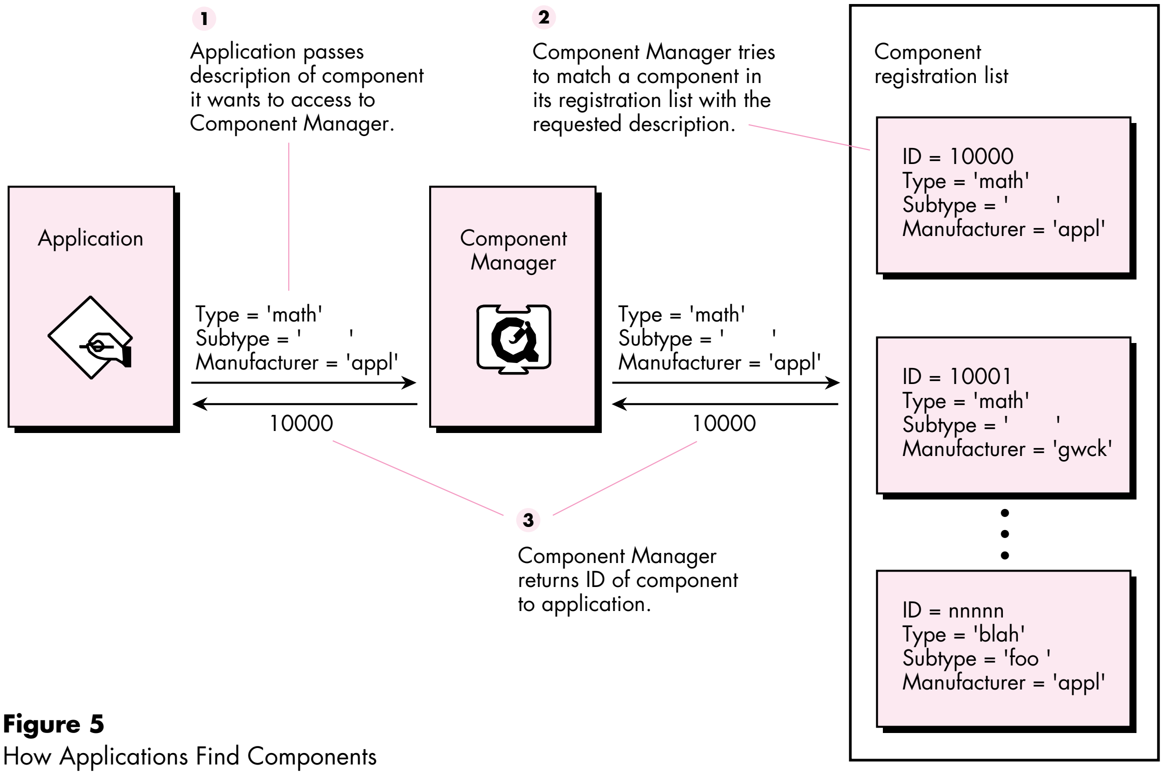


Figure 5
How Applications Find Components

HELPFUL TIP
You can obtain the component ID corresponding to a component instance by calling GetComponentInfo with the component instance (you'll need to cast it as a Component). The componentFlagsMask of the returned ComponentDescription record will contain the component ID.•

MAKING CALLS TO THE MATH COMPONENT

The Math component performs only two functions, dividing and multiplying two integers. To ask it to divide two numbers for us, we just call the component function DoDivide with the component instance value we got by opening the Math component.

```
result = DoDivide (mathInstance, numerator, denominator, &quotient);
```

When we're done with the component, we close the connection with the CloseComponent call, like this:

```
result = CloseComponent (mathInstance);
```

That's all there is to it. You can see that making component function calls is much like making any other kind of call.

EXTENDING EXISTING COMPONENTS

After defining the basic functionality for your component, you may find that you want to extend it beyond what you originally specified in your component API. There are three ways to extend the functionality of existing components:

- Use the subtype and/or manufacturer fields of the component description to indicate to a client application that a specific component implementation provides previously undefined functionality.
- Revise the component API to add calls that weren't specified in the original interface.
- Modify the behavior of a particular component implementation by capturing it and overriding a specific function.

The following sections examine these methods in detail.

ADDING NEW FUNCTIONALITY TO A SPECIFIC COMPONENT IMPLEMENTATION

Let's add some more functionality to the Math component. The MoMath component extends the Math component by adding an addition function. A new function prototype is added for the new function in MoMathComponent.h, along with a new request code, kDoAddSelect.

```
pascal ComponentResult DoAdd (MathComponent mathInstance, short firstNum,
   short secondNum, short* result) = ComponentCallNow (kDoAddSelect,
   0x08);
```

Request codes for implementation-specific functions must have an ID of 256 or greater. This is required to differentiate these functions from those that are generally defined in the API for the component type. Implementation-specific functions usually provide capabilities beyond those specified in the component API, and thus offer developers a way to differentiate their component implementations from those of competing developers. The following code fragment from the MoMath component dispatcher shows support for the DoAdd function:

```
case kDoAddSelect:         // Add function
{
   result = CallComponentFunction (params,
               (ComponentFunction) _MoMathDoAdd);
   break;
}
```

How does the calling application know that a superset of the Math component is around? To start with, the caller needs to know that such a beast even exists. Remember, this is an extension of a component implementation by a particular vendor, not of the component type in general. In this case, the extended component is differentiated from its basic implementation by its manufacturer code. Both Math and MoMath have the same component type ('math'), but their manufacturer codes differ ('appl' for Math and 'gwck' for MoMath). Note that the subtype field can be used in a similar manner, but it's typically used to distinguish algorithmic variations of a general component type. For example, image compressor components ('imco') use the subtype field to differentiate various types of compression algorithms ('rle ' for run length encoding, 'jpeg' for JPEG, and so on). The manufacturer field is used to identify vendor-specific implementations of a particular compression algorithm.

If the application is aware that this extended component exists, it can use the information stored in the component's 'thng' resource to locate and open it. Once the component has been opened, the application calls the extended function just as it would any other component function.

ADDING NEW FUNCTIONALITY TO A COMPONENT TYPE

In the preceding example, we used the manufacturer code to hook in new functionality to the Math component; this allowed a specific implementation to extend the interface. In reality, we would be better off extending the component by defining a change to the Math component API, so that all components of this type would have an interface defined for the new addition function. Of course, this is an option only when you're the owner of the component API. Changing component APIs that are owned by others (for instance, by Apple) is a good way to break applications, and no one appreciates that, least of all your users.

If you're going to take this route, be sure that the existing API is left unchanged, so that clients using the old component's API can use your new component without having to be modified. In addition, it's important to update the interface revision level of components that implement the new API, so that clients can determine whether a particular component implementation supports the new API.

MODIFYING EXISTING FUNCTIONALITY

Modifying existing functionality is a little more complicated than adding functionality to a component type. In the example component, the DoDivide function divides two 16-bit integers, truncating the result. We would actually get a better answer if the result were rounded to the nearest integer. We don't need to add a new call to do this, since what we really want to do is replace the implementation of the existing call with a more accurate version. On the other hand, the Math component does an acceptable job of multiplying two integers, so we don't need to override that function. Instead, we'll use the multiply function that's already implemented.

We can do this by writing a component that does the following:

- captures the original Math component
- overrides the original DoDivide function with a more accurate division function
- delegates the DoMultiply function to the original Math component

Let's start by writing a new component — in the example code, it's called NuMathComponent — that contains a dispatcher, as well as functions to handle the Component Manager request codes and the new DoDivide routine. We use a register routine to check for the availability of a Math component before we allow the NuMath component to be registered. If no Math component is available, obviously we can't capture it, and we shouldn't register. We also set cmpWantsRegisterMessage (bit 31) in the componentFlags field of the ComponentDescription record in the NuMath component's 'thng' resource to let the Component Manager know that we want a chance to check our environment before we're registered. With this flag set, the sequence of requests that NuMath will get at registration time will be open, register, and close.

The NuMath component register routine is as follows:

```
pascal ComponentResult _NuMathRegister (void)
{
   // See if a Math component is registered. If not, don't register
   // this component, since it can't work without the Math component.
   // We return 0 to register, 1 to not register.

   ComponentDescription mathDesc;
   mathDesc.componentType = mathComponentType;
   mathDesc.componentSubType = 0L;           // Wild card
   mathDesc.componentManufacturer = 'appl';
   mathDesc.componentFlags = 0L;             // Wild card
   mathDesc.componentFlagsMask = 0L;         // Wild card

   return ((FindNextComponent (nil, &mathDesc) != 0L) ? 0L : 1L);
}
```

Our open routine opens an instance of the Math component normally, and then uses the ComponentFunctionImplemented routine to determine whether the component we want to capture supports the target request code. We then capture the Math component with the CaptureComponent call.

```
if (ComponentFunctionImplemented ((ComponentInstance) mathComponentID,
      kComponentTargetSelect)) {
   mathComponentID = CaptureComponent (mathComponentID, (Component) self);
}
```

The original Math component ID is now effectively removed from the Component Manager's registration list. This means that the Math component is now hidden from all other clients, except those that already had a connection open to it before it was captured.

We then open an instance of the Math component, and use the ComponentSetTarget utility (defined in MathComponent.h) to inform Math that it's been captured by NuMath.

```
result = ComponentSetTarget (mathInstance, self);
```

Why does a component need to know that it's been captured? If a captured component makes use of its own functions, it needs to call through the capturing component instead of through itself, because the capturing component may be overriding one of the calls that the captured component is using. A captured component does this by keeping track of the component instance that the ComponentSetTarget call passed to it and by using that instance to make calls to the capturing component.

When the NuMath Comp;onent receives a divide request code, we dispatch to the new DoDivide function, effectively overriding the DoDivide function that was implemented in the Math component. However, when we receive a multiply request code, we delegate this to the captured Math component, since we aren't overriding the multiply function. We do this by simply making a DoMultiply call to the Math component, passing in the parameters that the NuMath component was provided with.

In our sample code, ComponentSetTarget is defined in MathComponent.h because the QuickTime 1.0 Components.h interface file doesn't declare it. The ComponentSetTarget declaration is included in newer QuickTime interface files, so if you're using them, you should comment it out in MathComponent.h.•

```
result = DoMultiply (mathInstance, firstNum, secondNum,
                        multiplicationResult);
```

In the close routine of the NuMath component, we remember to close the instance of the Math component we were using, and also to uncapture it so that we restore the system to its original state.

```
result = CloseComponent (mathInstance);
result = UncaptureComponent (mathComponentID);
```

THAT WASN'T SO BAD, WAS IT?

As you can see, adding new functionality is no big deal. As always, however, you should notify developers who may use your component of any late-breaking interface changes. You want to be sure that everyone's writing code that conforms to your most recent component specification.

ELEVEN COMMON MISTAKES

You may encounter some pitfalls during the development of your component. Here we discuss 11 common mistakes that we've either made personally or observed other developers make. We hope that you'll learn from our own fumblings and save yourself time and frustration.

Allocating space at registration time. Generally, it's best if your component allocates its storage only when it's about to be asked to do something — that is, when it has received a kOpenComponentSelect request code. This way, memory isn't tied up unnecessarily. Remember, your component may *never* be called during a given session, and if it's not, it shouldn't hang out sucking up memory some other process might be able to use.

Allocating space in the system heap. The system heap shouldn't be your first choice as a place to put your component globals. The system heap is generally reserved for system-wide resources (big surprise), and most components fall into the category of application resources that needn't be resident at all times. Consider carefully whether you need to scarf up system space. In addition, if your component is registered in an application heap, you should never try to allocate space in the system heap. The fact that you're registered in an application heap probably indicates that there isn't any more space in the system heap for you to grab.

Not supporting the kComponentVersionSelect request code. This is a pretty nasty omission for several reasons. First, this is the *easiest* request code to implement; it takes only a single line of code! What are you, lazy? (Don't answer that.) Second, clients may use the API version level to keep track of extended functionality — it may be that version 2 of a component interface contains additional calls over version 1, and a client certainly has reason to want to know that. Third, clients may use the component version to determine, for example, whether the component in question contains a recent bug fix.

Incorrectly calculating the parameter size for your component function prototype. If you do this, you'll probably notice it right after calling the offending component function, since your stack will be messed up by however many bytes you failed to calculate correctly. A common instance of this error occurs when calculating the space required by a function call that has char or Boolean parameters. Under certain circumstances, Boolean and char types are padded to two bytes when passed as function parameters.

To illustrate, we'll look at two example declarations. How many bytes of stack space need to be reserved for the parameters of the following function?

```
pascal ComponentResult I2CSendMessage (ComponentInstance ti,
   unsigned char slaveAddr, unsigned char *dataBuf, short byteCount)
```

The correct answer is eight bytes. The slaveAddr parameter is promoted to two bytes, the dataBuf pointer takes four bytes, and the byteCount takes two bytes. The rest of the declaration then takes the following form:

```
   = ComponentCallNow (kI2CSendMessageSelect, 0x08);
```

Let's look at the next example. How many bytes of stack space does this function require?

```
pascal ComponentResult MyFunction (ComponentInstance ti,
   Boolean aBoolean, char aChar, short *aPointer)
```

The correct answer is six bytes. The aBoolean parameter takes one byte, the aChar parameter takes one byte, and the aPointer parameter takes four bytes. What's that? Didn't we just say that Boolean and char parameters got padded to two bytes? We certainly did, but these types get padded only when an odd number of char or Boolean parameters occurs consecutively in the declaration. Because we could add one byte for the Boolean to the one byte for the char following it, we didn't need to do any padding — the total number of bytes was even (two bytes), and that's what's important. In the first example, this didn't work. We added one byte for the char to the four bytes for the pointer following it, and got five bytes, and so we needed to pad the char parameter by one byte. The rest of the declaration for the second example is

```
   = ComponentCallNow (kMyFunctionSelect, 0x06);
```

Registering your component when its required hardware isn't available. If your component doesn't depend on specific hardware functionality, don't worry about this. If it does (as, for example, video digitizers do), make sure you check for your hardware before you register your component. The Component Manager provides a flag, cmpWantsRegisterMessage, that you can set in the componentFlags field of your component description record to inform the Component Manager that your component wants to be called before it's registered. This gives your component an opportunity to check for its associated hardware, and to decline registration if the hardware isn't available.

Creating multiple instances in response to OpenComponent calls when your component doesn't support multiple instances. Only you can know whether your component can be opened multiple times. For instance, the Math component is capable of being opened as many times as memory allows (although our sample code restricts the number of open instances to three for the sake of illustration). Normally, a component that controls a single hardware resource should be opened only once and should fail on subsequent open requests. This will prevent clients from oversubscribing your component.

Not performing requisite housekeeping in response to a CloseComponent call. Bad things will happen, especially if you have hierarchies of components! As part of your close routine, remember to dispose of your private global storage and to close any drivers, components, files, and so on that you no longer need.

Allowing multiple instances from a single registration of a hardware component instead of allowing a single instance from each of multiple registrations. While this isn't really a common mistake today, we want to emphasize that there's a big difference between designing your component to allow multiple instances versus registering the component multiple times and allowing each registered component to open only once. In the case of a generic software library element (like Math), there's no problem with multiple instances being opened. In the case of a hardware resource that's being controlled with a component, it's almost always preferable to register the component once for every resource that's available (four widget cards would result in four different registrations rather than one registration that can be opened four times).

Why does it matter? Consider an application whose sole purpose in life is to manage components that control hardware resources. It may be selecting which resource to use, which one to configure, or which one to pipe into another. It's much more natural to ask the Component Manager to provide a list of all components of a certain type than it is to open each component that fits the criteria *n* times (until it returns an open error) in order to determine how many are available.

To kill a dead horse, suppose we have three identical video digitizers, and we want to convey that information to the user via a menu list. If all are registered separately, we can easily determine how many video digitizers are available (without even opening them) by using the FindNextComponent call. If only one were registered, the list presented to the user would only be a partial list. Take the blind leap of faith: register duplicate hardware resources!

As a final note, if you're registering a single component multiple times, be sure that the component name is unique for each registration. This allows users to distinguish between available components (as in the menu example in the previous paragraph), and it also helps you avoid the next gotcha.

Always counting on your component refCon being preserved. We know this may be upsetting to many of you, but there exists a situation in which your component refCon may not be valid. A component refCon (similar to a dialog, window, or control refCon) is a 4-byte value that a component or client can use for any purpose. It's accessed through a pair of Component Manager calls, GetComponentRefcon and SetComponentRefcon. Component refCons are frequently used to hold useful information such as device IDs or other shared global data, and so can be quite critical to a component. We can hear you now . . . "*What*? You're going to nuke my *global data* reference?!" Well, not exactly — it's just not as immediately accessible as you would like it to be. Don't worry, it's possible to detect when your component is in this situation and retrieve the refCon from it, as long as you follow a few simple steps.

The situation in question arises when there's not enough room in the system heap to open a registered component. This happens when you run an application (that uses your component) in a partition space so large that all free memory is reserved by the application. This will prevent the system heap from being able to grow. When the application calls OpenComponent, the component may be unable to open in the system heap because there's no available space. In this case, the Component Manager will *clone* the component. When a component is cloned, a new registration of the component is created in the caller's heap, and the component ID of the cloned component is returned to the caller, *not* the component ID of the original registration. The clone is very nearly a perfect copy, but like the Dopplegänger Captain Kirk in the *Star Trek* episode "What Are Little Girls Made Of?" it's missing something crucial.

That something is the component refCon. The refCon isn't preserved in the clone, so if your component needs the refCon to perform properly, it must be recovered from the original component. How you go about doing this is a bit tricky. We assume that you followed our advice and made sure that your component registered itself with a unique name. (This technique is *not* guaranteed to work properly unless this constraint is satisfied — you'll see why shortly.)

The first problem is detecting whether your component has been cloned at open time. You can determine this by examining your component's A5 world using the GetComponentInstanceA5 routine. If the A5 world is nonzero, you've been cloned. But wait, you say, what if I registered my component locally? Won't it have a valid A5 value? Yep, it sure will, but if it was registered locally, we won't have this nasty situation to begin with, since the component won't be in the system heap anyway.

Now you know that you've been cloned, and that you can't depend on your refCon. How do you retrieve it? Well, we know that there are two registrations of the same component in the Component Manager registration list (the original and the clone). So all we have to do is to set up a component description for our component, and then use FindNextComponent to iterate through all registrations of it. We know what our current component description and ID are, so we can just examine the component description and ID for each component returned. Once we find a component whose ID is different from ours but whose description is identical, we've found the original component registration. We can then make a call to GetComponentRefcon to obtain the original refCon value, and then set the clone's refCon appropriately. Whew!

This technique won't work with a component that registers multiple times and doesn't register each time with a unique name. If component X, capable of multiple registrations, always registers with the name "X," then when we try to find the original component from the clone, there will be multiple components named "X" in the registration list, and we'll be unable to determine which component is the one we were cloned from.

Omitting the "pascal" keyword from declarations for your component dispatcher or for any functions that are called by CallComponentFunction or CallComponentFunctionWithStorage. This bug will only antagonize those developers who are working in C. As many of you know, the Macintosh system software was originally written in Pascal, and functions that are called by Toolbox routines (in this case, by the Component Manager) must conform to Pascal calling conventions. If you fail to include this keyword where necessary, the parameters for your function will be interpreted in the reverse order from what you intended, and your component may enter the Twilight Zone, perhaps never to return.

Trying to read resources from your component file when its resource fork isn't open. When one of your component functions is called, the current resource file (as obtained from CurResFile) is *not* the component's resource file unless you explicitly make it so. If you need to access resources that are stored in your component file, you must first call OpenComponentResFile to get an access path, and then call UseResFile with that path. When you're done with the file, restore the current resource file and call CloseComponentResFile to close your component file.

DEBUGGING TOOLS AND TECHNIQUES

Debugging components can be frustrating if all you have to work with is MacsBug. Fortunately, there are a few tricks and tools that give you a little more power to terminate those pesky bugs. In this section, we'll show you how to debug your component code with a symbolic debugger, and then we'll examine three utilities that will help you test your component.

SYMBOLIC DEBUGGING

Let's suppose that we've got the Math component up and running, but something funny is happening in our DoDivide routine. It would be nice to be able to step through the component code symbolically and see what's happening. Fortunately, there's a simple trick that involves registering our component in such a way that it can be symbolically debugged.

For the purposes of the example, we'll discuss how to do this with Symantec's THINK C development system. The first step is to add the component source code to the application source code project. Then we modify the application code so that instead of using the FindNextComponent call to locate the Math component, we register it ourselves using the RegisterComponent call.

```
#define kRegisterLocally 0
mathComponentID = RegisterComponent (&mathDesc,
   (ComponentRoutine) MathDispatcher, kRegisterLocally, nil, nil, nil);
```

Note that when you register a component in an application heap as we're doing, you must register it locally, or your system may die a horrible death after your application quits and its application heap goes away.

The component description, mathDesc, is set up just as before. The second parameter is the main entry point (the dispatcher) to the Math component. The Component Manager will call this routine every time it receives a request code for an instance of the Math component.

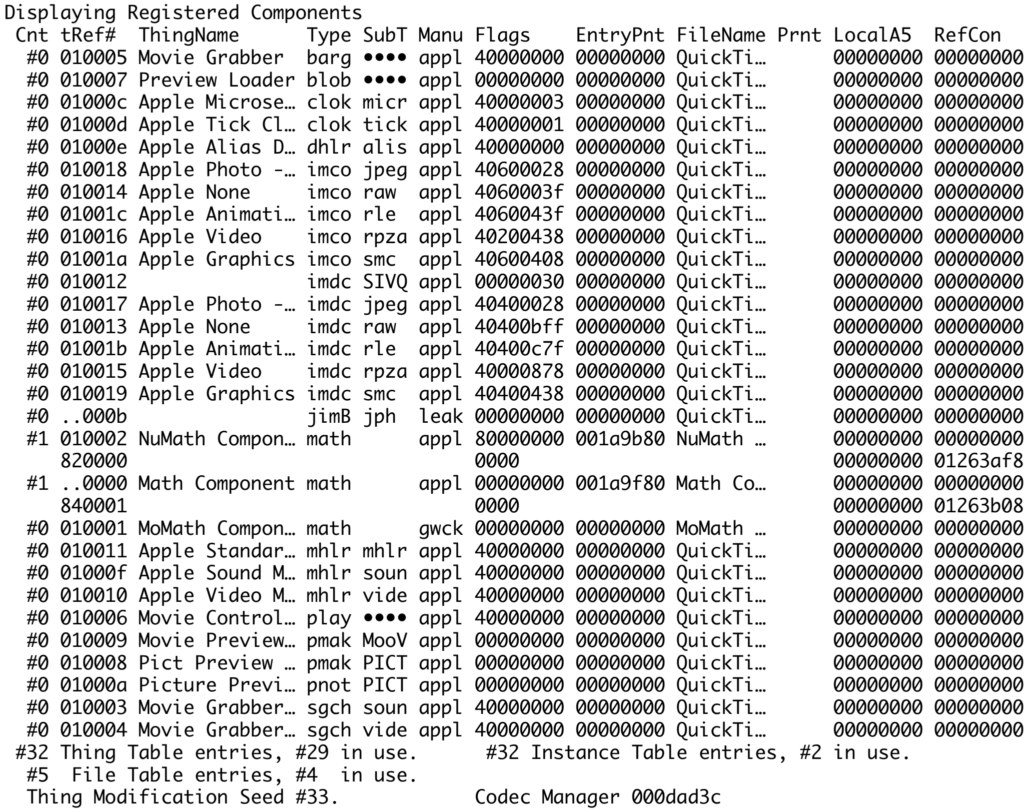
In the Math component code, we set up a debug compiler flag (DEBUG_IT, found in DebugFlags.h) which, if defined, indicates whether we want to declare our component dispatcher as a main entry point for a standalone code resource or as just another routine linked into our application program.

```
#ifdef DEBUG_IT
   // Use this declaration when we're running linked.
   pascal ComponentResult MathDispatcher (ComponentParameters *params,
                                              Handle storage)
#else
   // Use this declaration when we're building a standalone component.
   pascal ComponentResult main (ComponentParameters *params,
                                     Handle storage)
#endif DEBUG_IT
```

The two declarations differ only in that one is declared as a main and one isn't. (Remember, with both the source for the component and the application in the same project, we can't have two mains.) Now, each time the Component Manager sends a request code to the Math component, it's calling a component routine linked into the application (MathDispatcher) that we can trace with the debugger. When we've finished debugging the component, we can undefine the debug flag and rebuild the component as a standalone code resource. The test application will now use FindNextComponent to access the standalone component.

THE THING MACSBUG DCMD

The **thing** dcmd is included on the QuickTime 1.0 Developer's CD. To use this dcmd, simply use ResEdit to copy the 'thng' dcmd resource into a file named Debugger Prefs, and put this file into your System Folder. Once in MacsBug, the dcmd is invoked by entering "thing". A sample **thing** display is shown in Figure 6.



```
Displaying Registered Components
 Cnt tRef#  ThingName      Type SubT Manu Flags    EntryPnt FileName Prnt LocalA5  RefCon
  #0 010005 Movie Grabber  barg •••• appl 40000000 00000000 QuickTi…      00000000 00000000
  #0 010007 Preview Loader blob •••• appl 00000000 00000000 QuickTi…      00000000 00000000
  #0 01000c Apple Microse… clok micr appl 40000003 00000000 QuickTi…      00000000 00000000
  #0 01000d Apple Tick Cl… clok tick appl 40000001 00000000 QuickTi…      00000000 00000000
  #0 01000e Apple Alias D… dhlr alis appl 40000000 00000000 QuickTi…      00000000 00000000
  #0 010018 Apple Photo -… imco jpeg appl 40600028 00000000 QuickTi…      00000000 00000000
  #0 010014 Apple None     imco raw  appl 4060003f 00000000 QuickTi…      00000000 00000000
  #0 01001c Apple Animati… imco rle  appl 4060043f 00000000 QuickTi…      00000000 00000000
  #0 010016 Apple Video    imco rpza appl 40200438 00000000 QuickTi…      00000000 00000000
  #0 01001a Apple Graphics imco smc  appl 40600408 00000000 QuickTi…      00000000 00000000
  #0 010012                imdc SIVQ appl 00000030 00000000 QuickTi…      00000000 00000000
  #0 010017 Apple Photo -… imdc jpeg appl 40400028 00000000 QuickTi…      00000000 00000000
  #0 010013 Apple None     imdc raw  appl 40400bff 00000000 QuickTi…      00000000 00000000
  #0 01001b Apple Animati… imdc rle  appl 40400c7f 00000000 QuickTi…      00000000 00000000
  #0 010015 Apple Video    imdc rpza appl 40000878 00000000 QuickTi…      00000000 00000000
  #0 010019 Apple Graphics imdc smc  appl 40400438 00000000 QuickTi…      00000000 00000000
  #0 ..000b                jimB jph  leak 00000000 00000000 QuickTi…      00000000 00000000
  #1 010002 NuMath Compon… math      appl 80000000 001a9b80 NuMath …      00000000 00000000
     820000                               0000                            00000000 01263af8
  #1 ..0000 Math Component math      appl 00000000 001a9f80 Math Co…      00000000 00000000
     840001                               0000                            00000000 01263b08
  #0 010001 MoMath Compon… math      gwck 00000000 00000000 MoMath …      00000000 00000000
  #0 010011 Apple Standar… mhlr mhlr appl 40000000 00000000 QuickTi…      00000000 00000000
  #0 01000f Apple Sound M… mhlr soun appl 40000000 00000000 QuickTi…      00000000 00000000
  #0 010010 Apple Video M… mhlr vide appl 40000000 00000000 QuickTi…      00000000 00000000
  #0 010006 Movie Control… play •••• appl 40000000 00000000 QuickTi…      00000000 00000000
  #0 010009 Movie Preview… pmak MooV appl 00000000 00000000 QuickTi…      00000000 00000000
  #0 010008 Pict Preview … pmak PICT appl 00000000 00000000 QuickTi…      00000000 00000000
  #0 01000a Picture Previ… pnot PICT appl 00000000 00000000 QuickTi…      00000000 00000000
  #0 010003 Movie Grabber… sgch soun appl 40000000 00000000 QuickTi…      00000000 00000000
  #0 010004 Movie Grabber… sgch vide appl 40000000 00000000 QuickTi…      00000000 00000000
 #32 Thing Table entries, #29 in use.      #32 Instance Table entries, #2 in use.
  #5  File Table entries, #4  in use.
  Thing Modification Seed #33.            Codec Manager 000dad3c
```

Figure 6
Sample **thing** MacsBug Display

The Cnt field indicates the number of instances of a particular component.

The tRef# field shows the component ID that the Component Manager has assigned to a particular component; this is the value that's returned to your application by the FindNextComponent call. If there are instances of a component open, the component instances are listed below the component ID in the tRef# field. Note that the tRef# for the Math component is ..0000. The two dots at the beginning indicate that this component has been captured. (We know from the earlier discussion of the NuMath component that it has captured the Math component.)

The ThingName field displays the name of a particular component. This is either the string that's pointed to by the component's 'thng' resource or the name that it was registered with by a call to RegisterComponent.

The Type, SubT, Manu, and Flags fields likewise correspond either to the information that's stored in the component's 'thng' resource or to the codes and flags that were supplied to a call to RegisterComponent.

The EntryPnt field is the main entry point of the component code.

The FileName field indicates what file the component's 'thng' resource resides in. This field is empty for components registered without a component resource.

The Prnt field displays the parent of a cloned component. This information isn't available through the Component Manager API.

The LocalA5 field shows the A5 world that the component is associated with; unless the component is cloned or registered locally, this value is 0.

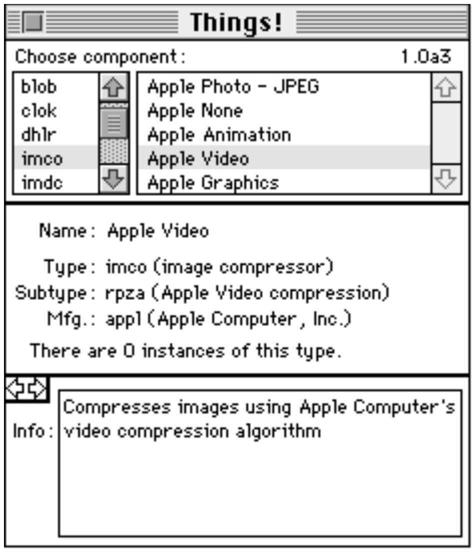
The RefCon field is the value of the component's refCon.

At the bottom of the display there's a decimal number indicating the number of component (thing) entries allocated in the Component Manager registration list, along with the number of entries actually in use. Similar information is given for the number of file table entries. Finally, the Component Manager modification seed is listed.

THINGS! CONTROL PANEL

The Things! control panel, included on the QuickTime 1.0 Developer's CD, is similar to the **thing** dcmd but provides several additional capabilities. These include displays of version levels, info and name strings, and resource information, as well as controls to reorder the component search chain and to unregister components.

Figure 7 shows a sample display of the Things! control panel.



Name: Apple Video

Type: imco (image compressor)
Subtype: rpza (Apple Video compression)
Mfg.: appl (Apple Computer, Inc.)

There are 0 instances of this type.

Info: Compresses images using Apple Computer's video compression algorithm

Figure 7
Things! Control Panel Main Display

The list on the left in the top panel shows the types of components currently registered with the Component Manager; the list on the right shows the components of the selected type that are currently registered. The latest version of Things! doesn't display components that aren't registered globally or that aren't registered in the same application heap as the control panel is operating in. Things! also doesn't show components that aren't resource-based.

The middle panel shows the name of the currently selected component and a description of its type, subtype, and manufacturer fields. The number of instances of the type of component selected (in the example, the 'imco', or image compressor, component type) is displayed at the bottom of this panel. Clicking this field will toggle it to display the number of instances of the selected component (in this case, the Apple Video image compressor component).

The bottom panel shows an information string that usually describes what the component does. At the upper left in this panel are two arrow buttons that can be used for paging the bottom panel (the top and middle panels don't change).

Figure 8 shows a variation of the bottom panel's second page. The component version information is displayed at the top. The "Set default" button allows you to assign a particular component as the first component in the Component Manager's search chain for that component type.

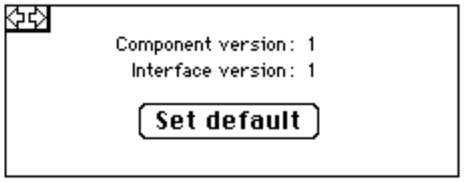

Component version: 1
Interface version: 1

Figure 8
Things! Page 2 Display

If the Option key is held down while paging to the second page, a Destroy button is displayed (as shown in Figure 9). Clicking this button will unconditionally unregister the currently selected component.

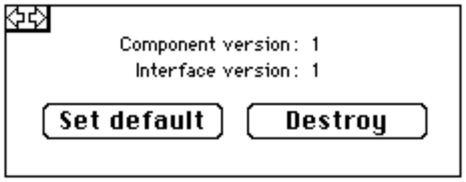

Component version: 1
Interface version: 1

Figure 9
Things! Extended Page 2 Display

The third page shows the flags and mask fields of the component.

The fourth page displays a variety of information about the 'thng' resource associated with a particular component, including the resource name and ID as well as its attributes.

Page 5 presents a summary of the system software configuration.

REINSTALLER
Reinstaller is a utility that lets you install resource-based components without restarting your Macintosh. Launching the application presents a Standard File dialog asking you to choose the file containing the component you want to register. Clicking the Open button will dismiss the dialog and register the selected component with the Component Manager.

The same component file can be installed multiple times. Duplicate components aren’t removed; the most recently installed version of a component becomes the default component for that type. Note that any components installed with Reinstaller are installed only until shutdown or reboot.

This utility is quite handy in conjunction with the Things! control panel's Destroy button. Between the two of them, you can easily register and unregister your components without having to restart your Macintosh.

GO DO YOUR OWN "THING"

Now you know how easy it is to write your own components. You've learned how to declare your own component API and how to implement a component dispatcher for it. You've seen what common pitfalls to avoid and how to symbolically debug your component to help you get around new pitfalls we haven't thought of.

We're confident that once you start programming components, you'll become addicted! So what are you hanging around here for? Get busy writing, and start amazing your users (and us, too) with some way cool components. We're waiting . . .

REQUIRED READING

- *QuickTime Developer's Guide* (part of the QuickTime Developer's Kit v. 1.0, ADPA #R0147LL/A). Currently the essential reference for programming with the Component Manager. This documentation will be replaced in the near future by three new *Inside Macintosh* volumes: *QuickTime, QuickTime Components*, and *More Macintosh Toolbox.* (The Component Manager will be documented in the latter volume.)
- "QuickTime 1.0: 'You Oughta Be in Pictures,'" Guillermo A. Ortiz, *develop* Issue 7. An overview of QuickTime, including the Component Manager.

THANKS TO OUR TECHNICAL REVIEWERS
Neil Day, Mike Dodd, Mark Krueger, John Wang•

DAVID VAN BRINK

BE OUR GUEST

COMPONENTS AND C++ CLASSES COMPARED

If you're familiar with C++ classes but new to thinking about components, you may find it instructive to know how the two compare. Although each has its own niche in Macintosh software development, components and C++ classes have many features in common.

In general, both components and C++ classes encourage a building-block approach to solving complex problems. But whereas a component is separate from any application that uses it, a class exists only within the application that uses it. Components are intended to add systemwide functionality, while classes are intended to promote a modular approach to developing a program.

We can also compare components and C++ classes in terms of how they're declared and called, their use of data hiding and inheritance, and their implementation. But first, let's briefly review what a class is and what a component is.

SOME BASIC DEFINITIONS

A class, in the programming language C++, is a description of a data structure and the operations (methods) that can be performed on it. An instance of a class is known as an object. Classes are provided in C++ to promote an "object-oriented programming style." By grouping a data type and its methods together, classes enable programmers to take a modular approach to developing a program.

A component, as described in the preceding article ("Techniques for Writing and Debugging Components"), is a single routine that accepts as arguments a selector and a parameter block. The selector specifies which of several (or many) operations to perform, and the parameter block contains the arguments necessary for that operation. Components are "registered" with the Component Manager and can be made available to either the program that registered the component or to any program that's executed, making it possible to add systemwide functionality. For instance, if Joe's Graphics Corporation develops a new image compression technique, it can be sold to users as a component. Users install the component simply by dragging an icon into a folder, and that form of image compression is then automatically available to all programs that make use of graphics.

DECLARING CLASSES AND COMPONENTS

A C++ class is declared in much the same way as a struct, with the addition of routines that operate only on the structure described. Once the class is declared, instances can be declared in exactly the same way as other variables. That is, to create an instance of a class, you either declare a variable of that class or dynamically allocate (and later deallocate) a variable of that class.

A component must be registered with the Component Manager. At that time, its type, subtype, manufacturer, and name are specified. The type, subtype, and manufacturer are long integers; the name is a string.

Component instances can only be created dynamically, using specific Component Manager routines. To create an instance of a component that has been registered, a program must first find the component. If the seeking program is the same one that registered the component, it already has the component. If not, it can make Component Manager calls to search for all available components with a given type, subtype, and manufacturer; any part of the description can be a wild card.

Once a component has been found, it must be opened, and this operation produces a reference to the

DAVID VAN BRINK is a computer programmer. When he's not busy programming computers, he can usually be found writing computer programs. Mostly, he does this in the soothing fluorescent glow of his cubicle at Apple. He's presently writing components (with great fervor) to support musical synthesizers for QuickTime.•

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.•

component instance. Operations can be performed on the component instance using this reference.

Table 1 compares how classes and components are declared and how instances of each are created. (Note that for components, the code is idealized.)

CALLING ALL ROUTINES

Calling a routine that operates on a C++ object is slightly different from making a standard routine call: the call more closely resembles a reference to an internal field of a struct. The routine that gets called is identical to any other routine, except that it's declared within the class definition rather than at the same brace level as the main routine.

Calling a component routine is identical to calling any other routine. The first argument is always the component instance, and other arguments may optionally follow. The return type of every component routine is a long integer, and part of the numerical range is reserved for error messages from either the component or the component dispatch mechanism.

The Component Manager lets a program issue calls to a component that it has never "met" before. This form of dynamic linking is crude, because no type checking is performed.

Table 1 compares how classes and components are called.

DATA HIDING

A C++ class can have "private" fields and methods, which are accessible by class methods but not by the caller. The programmer can see these private parts simply by perusing the class declaration. If a change to the implementation of a class requires that the private parts be changed, relinking with the implementation of the class won't be sufficient: all clients must be recompiled, since the positions of public fields might have changed. (One tricky way around this is to include a private field of type char * that's really a pointer to the class's internal state data. The class constructor allocates memory for whatever internal state it likes and coerces a pointer to it to live in that char * field. This technique is useful for object-only software library distribution and also protects proprietary algorithms from curious programmers.)

A component is responsible for allocating memory for its internal state (the component's "globals") when it's opened and releasing that memory when it's closed. There are both component globals and component instance globals. These correspond to static and automatic variables in a C++ class and have similar utility. A component might keep track of how many instances of itself have been opened and restrict that number by failing on the open call.

INHERITANCE

It's often useful to build software on top of existing functionality or, alternatively, to take existing functionality and alter it to perform a more specialized function. Both of these things can be accomplished for C++ classes with inheritance. In the former case, the new class will have methods that don't exist in the base class; in the latter, the new class will have methods with the same name as methods in the base class but that take precedence over the base methods.

Components and the Component Manager support both kinds of inheritance as well, as discussed in the preceding article. All components of a given type must support the same set of calls, although this is enforced only by convention. Components of a particular type and subtype may optionally support other calls as well, and components of a particular type, subtype, and manufacturer may support still more calls.

In the case where a component wants to use the services of another component and perhaps override some of its functions with modifications, Component Manager utilities let a component designate another component as its "parent." A simple protocol ensures that the correct variant of a routine gets called. When a component must call itself, it must issue the call to its child component, if any. When a component wants to

Table 1
A Comparison of Calls: Classes (Actual Code) Versus Components (Idealized Code)

Declaring a Class

```
class MyClass {
/* Variables and methods for
   the class */
}
```

Declaring a Component

```
myComponent = RegisterComponent(MyEntryRoutine,
      myType, mySubType, myManufacturer,
      "A Component");
```

Creating a Class Instance

```
MyClass x;
```

Creating a Component Instance

```
myComponent = FindComponent(myType, mySubType,
   myManufacturer);
myInstance = OpenComponent(myComponent);
```

Calling a Class

```
x.MyMethod(arg1, arg2);
```

Calling a Component

```
result = MyMethod(myInstance, arg1, arg2);
```

Implementing a Class

```
class MyClass {
   void MyMethod(int arg1, int arg2)  {
   /* Some code for MyMethod */
   }
}
```

Implementing a Component

```
long MyEntryRoutine(ComponentParams *params,
                      char *globals)  {
   switch(params->selector)  {
      case kOpen:
      case kClose:
         return noErr;
      . . .   /* other required calls here */
      case MyMethod:
         /* Do my method. */
         /* arg1 and arg2 are in params. */
         return noErr;
      default:
         return routineNotImplementedErr;
   }
}
```

rely on the existing implementation of the parent component, it must pass the call to its parent.

IMPLEMENTING CLASSES AND COMPONENTS

My discussion of implementation is based on the 68000 platform, since that's the only one I've scrutinized with regard to compiled C++ and Component Manager calls.

The routines that can be used with a C++ class are declared, and optionally implemented, within the class declaration. They behave like normal C routines, as described earlier.

A call to a C++ class that has no parents or descendants is compiled as a direct subroutine call, exactly as is a standard routine call. A call to a C++ class that has parents or descendants is slightly more complicated. A table lookup is used at run time to determine exactly which implementation of a routine gets called for the particular object being operated on. Such a call takes perhaps a dozen assembly instructions.

A component consists of only a single routine. It's passed a selector and a parameter block. The selector is used to decide which operation to actually perform, and the parameter block contains all the arguments passed by the caller.

The component's parameter block is untyped — the component routine has no way to determine what kinds of arguments were originally passed, and herein lies the danger. Some languages, such as LISP, have untyped arguments; in LISP, however, a routine can determine how many arguments have been passed and what the argument types are. A component interface is more like assembly language — or C without prototypes! — in that it can determine nothing about what has been passed to it.

You can't compile a C++ program containing a call to a nonexistent routine; the compiler will balk. (Well, OK, this isn't strictly true: there are dynamically linking systems for C++, and other languages, that let you call a C++ routine that hasn't been linked with the rest of the compiled source code; the routine can be linked to later, at run time. But no facility of this type is currently standard in the Macintosh Operating System or supported under the standard Macintosh development tools.) In the case of components, the compiler can't check for such illegal calls, since the particular components that may be opened are decided at run time. Therefore, the caller must be prepared to handle a "Routine Not Implemented" error if a call is made with an unknown selector.

All calls to components pass through the Component Manager's dispatch mechanism. The dispatcher must locate the component's entry point and globals from the component reference, which is not simply a pointer but a packed record containing an index into a table and some bits used to determine whether the component reference is still valid. If a client makes a call to a component it no longer has open, the Component Manager has a statistical likelihood of catching this call and returning an appropriate error.

The Component Manager has facilities to redispatch the parameter block to one of many routines, and those routines are written to take the arguments as originally passed. The Component Manager was originally written for use on the 68000 series of processor; on computers with that processor, the parameter block doesn't have to be recopied onto the stack for further dispatching. On other processors the parameters might have to be recopied, however.

The Component Manager has been highly optimized and fast dispatching can reduce its overhead still more, but in general its lookup-and-dispatch process still takes several dozen instructions. If the component being called is using the Component Manager's inheritance mechanism, further overhead is incurred by passing control to the parent or child component. Overall, the Component Manager is quite efficient, but still not as efficient as direct routine calls.

Table 1 compares how classes and components are implemented.

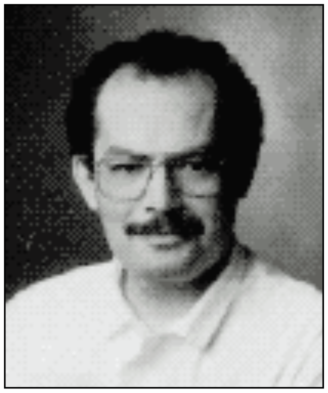
IN SUM

Components, as supported by the Component Manager, exhibit many of the features of C++ classes. Both encourage a modular approach to solving problems. Both feature inheritance and data hiding. Where they differ is in how they're declared and implemented, how they do (or fail to do) type checking, and how expensive they are to call. Each occupies its own distinct niche in Macintosh programming: classes as a way to ease development of a single program, components as a way to add systemwide functionality and give control and choice to the user.

Thanks to Casey King and Gary Woodcock for reviewing this column.•

TIME BASES: THE HEARTBEAT OF QUICKTIME

A time base is the heartbeat of a QuickTime movie. It keeps the movie going and tells the Movie Toolbox when to stop and when to display the next frame. This article explores the concept of time bases and shows how you can use time bases to affect the behavior of movies as well as how to use time base callback procedures for timing purposes.



GUILLERMO A. ORTIZ

In a basic sense, a time base can be viewed as the black box that maintains the temporal characteristics of a QuickTime movie. When a movie is playing, some of its temporal characteristics are obvious: it has a duration, it's "moving" or not, and so on. Some of the not-so-obvious characteristics that a time base keeps track of are also important, such as the clock that governs the passing of time as far as the movie is concerned and whether the movie is looping.

Time bases are created dynamically by the Movie Toolbox each time a movie is opened, rather than being stored with a movie. Time bases can also exist by themselves, with no movie attached, and can therefore be used to time other dynamic events, much as you would use a Time Manager task.

The QuickTime Movie Toolbox provides a high-level interface that lets you modify all the parameters of a movie, some of which implicitly change its associated time base. Most applications therefore will never need to manipulate time bases directly. Nevertheless, there are situations in which it's necessary to access time bases more directly, such as the following:

- when a document presents multiple views of a movie and all views need to be kept in sync
- when you need to take advantage of the callback functions of a time base
- when you're writing a custom movie controller

This article addresses these situations.

GUILLERMO A. ORTIZ When I met Guillermo I was really young. The first thing I said to him was, "Are you my dad?" Well, really I said something like "Waaaa!" As a matter of fact, I was a newborn. What I like about Guillermo is that we are almost the same. We like the same food and we watch the same TV shows, like *Star Trek* and *Nova*. We like to play the same sports, such as tennis and basketball, and we both like to read a lot — lately we even read the same books. But the one way we are most alike is that we love computers. Isn't that a weird coincidence? Another thing I like about Guillermo is that he is really smart. He used to be a teacher, and having a teacher around the house is always useful. He is always there to help me with stuff like math and science. He is also a great cook. The dinners he prepares for us on the weekends are fit for a

THE ARROW OF TIME

First let's define some of the terms related to the way QuickTime treats time:

- Time scale: the number of units into which a second is subdivided. For most QuickTime movies, the time scale is set to 600, meaning that the smallest fraction of time measurement for the movie is 1/600th of a second.
- Rate: the multiplier for the time scale. The rate controls how fast the movie plays. When the rate is 1.0, the movie plays at its normal speed, meaning that for each second of play the movie's time advances a number of units equal to the time scale. If the rate is between 0.0 and 1.0, the movie plays in slow motion, and fewer units are counted off per second of play. A negative rate implies that the movie is playing backward. A rate of 0 means that the movie is stopped.
- Time value: indicates where we are in the movie being played back. The time value is given as a number of time scale units. When a movie is playing forward from its start, the current time value can be calculated as

  time elapsed (in seconds) * time scale * rate

You can think of a time base as a dynamic container that holds the following information about a process, normally a movie: the time source (either the clock being used as the master clock or the master time base); the time bounds (the start and stop time values) and the current time value; the rate; the time base flags, indicating different conditions for the time base; and a list of callback routines.

Figure 1 illustrates these concepts and shows how they interact. The figure assumes that the clock or time source ticks at a constant speed; however, you could conceivably use a clock that runs at a varied speed, which would make the movie go faster and slower in sync with the clock.

Figure 1 doesn't show the effect of the time base flags. In QuickTime versions 1.0 and 1.5, two mutually exclusive flags are defined — loopTimeBase and palindromeLoopTimeBase. The loopTimeBase flag causes the time base to go back to the start time value when it reaches the stop time value (or vice versa if the movie is playing in reverse); palindromeLoopTimeBase reverses the direction of play when it gets to the start or stop value of the time base.

THE BASIC STUFF

The QuickTime Movie Toolbox is the best mechanism for manipulating movies and their parameters. The high-level calls provided by the Toolbox are all that most applications will ever need. Using graphics as an analogy, suppose that you wanted to

king! He is like my family's own personal four-star chef. The music Guillermo likes to listen to is the Beatles, the Doors, Santana, and Cream. I like those groups too, but on our way to school we listen to rap. Guillermo is a great guy and I am really glad to have him for a dad.

— Guillermo A. Ortiz Jr., age 13•

RECOMMENDED READING

This article assumes a reasonable knowledge of QuickTime programming. For background information, see the QuickTime developer notes, available from APDA as part of the QuickTime Developer's Kit v. 1.0 (#R0147LL/A). For an introduction to QuickTime, see my article "QuickTime 1.0: 'You Oughta Be in Pictures'" in *develop* Issue 7.•

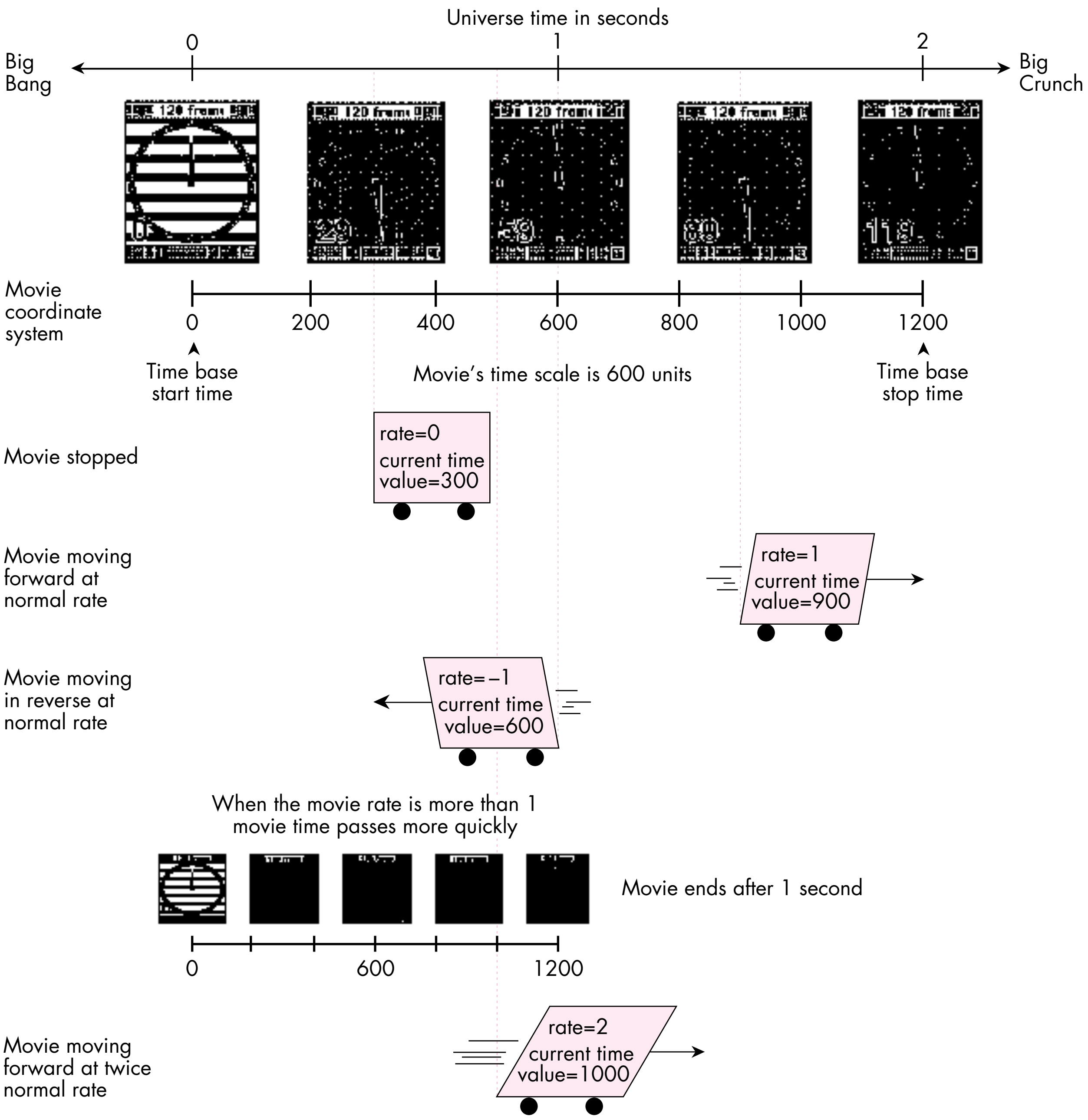
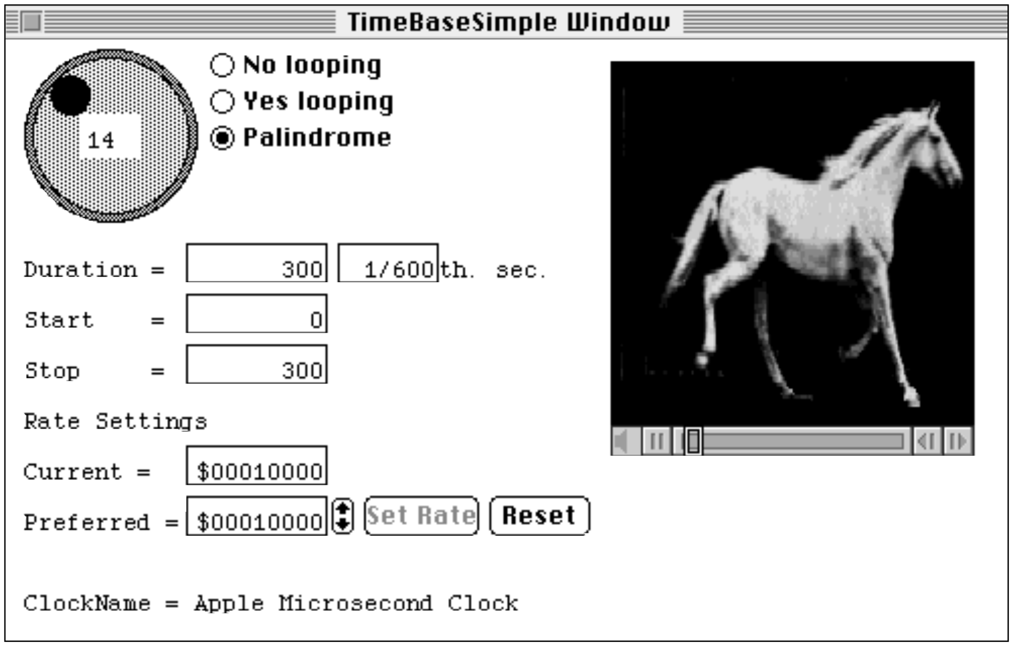


Figure 1
Time Concepts in a QuickTime Movie

draw a complicated image. The easiest way to do this would be with QuickDraw, by calling DrawPicture, but you could also interpret the picture by hand and execute its opcodes individually or even set the video RAM pixels directly! Similarly, when working with a movie, you can work directly with its time base, but it's best to let the Movie Toolbox do as much as possible — not because it's the only way, but because it's safer, it lessens the chances for compatibility problems, and it's simpler. Thus, for time bases associated with movies, it's best to call SetMovieTime rather than SetTimeBaseTime and to call SetMovieMasterClock rather than SetTimeBaseMasterClock.

For those cases in which it makes sense to access and modify time bases directly (as in the scenarios mentioned earlier), the Movie Toolbox provides procedural interfaces that allow you to do so. The sample program TimeBaseSimple provided on the *Developer CD Series* disc shows how to get a time base from a movie, how to interrogate the time base, and how to change some of its parameters.

Figure 2 shows the window displayed by TimeBaseSimple. This window displays the duration of the time base (in most cases the same as the duration of the movie), a number obtained by subtracting the start time value from the stop time value. It also shows the rate at which the movie is playing, the preferred rate (a value normally set



TimeBaseSimple Window

○ No looping
○ Yes looping
● Palindrome

Duration = 300 1/600th. sec.
Start = 0
Stop = 300

Rate Settings
Current = $00010000
Preferred = $00010000

ClockName = Apple Microsecond Clock

Figure 2
TimeBaseSimple Window

when the movie is created; it will differ from the rate if the movie is stopped or is playing in reverse due to palindrome looping), the name of the clock being used, and the type of looping in effect.

Through this window, the user can set the preferred rate, which is the rate the Movie Toolbox and the standard movie controller use to set the movie in motion. Radio buttons allow the user to specify the type of looping via the time base flags. The user can also scan the movie backward and forward by clicking the shuttle control in the top left of the window. This control is included in the sample to show how to go forward or backward in the movie by changing the current time value in the movie's time base.

GETTING AND CHANGING A TIME BASE

Before you can begin working with a time base, you have to get it. TimeBaseSimple does this with the following line:

```
tb := GetMovieTimeBase(gMoov);         (* get movie's time base *)
```

GetMovieTimeBase returns the time base associated with the movie gMoov. The variable tb receives this value.

Getting the clock information. Once you've retrieved the time base, you can get the information about it. TimeBaseSimple acquires the information regarding the master clock in order to display its name in the window. The clock information is obtained via the Component Manager. First we obtain the clock component being used by the time base; then we use it to get the information from the Component Manager.

```
clock := GetTimeBaseMasterClock(tb);   (* instance of clock being used *)
err := GetComponentInfo(Component(clock), cd, Handle(clockN), NIL, NIL);
```

In the variable cd, a ComponentDescription record, GetComponentInfo returns the type of component, the subtype, the manufacturer, and the component flags. Note that the program could be written to pass NIL instead, because the information received is not used. clockN is a handle in which GetComponentInfo returns the name of the component, which is what we're really looking for.

Note also that when a time base has been enslaved to another (as discussed later), GetTimeBaseMasterClock returns NIL. To ensure there's a master clock associated with a time base, the application should first call GetTimeBaseMasterTimeBase; a NIL result indicates that the time base has a master clock, whereas a nonzero result indicates that a master time base exists that contains the master clock.

Getting and changing the time values. You can get the start and stop time values for a time base as follows:

There are no guidelines (as of this writing) regarding ways to allow users to set the rate of a movie; the solution implemented in TimeBaseSimple has not been approved by the User Interface gods. It's primitive, but it works.•

The TimeBaseSimple shuttle control is a cool CDEF created by C. K. Haun, member emeritus of Developer Technical Support. Its C source is included on the *Developer CD Series* disc.•

```
scale := GetMovieTimeScale(gMoov);  (* first get the time scale *)
startTimeValue :=
   GetTimeBaseStartTime(tb, scale, startTime);  (* get start time *)
stopTimeValue :=
   GetTimeBaseStopTime(tb, scale, stopTime);    (* get stop time *)
```

Note that the start and stop times returned are given in terms of the time scale being passed; this means that we can get different values for the same time point, depending on the granularity we require. As a matter of fact, in TimeBaseSimple, when we're preparing the shuttle control, we get the same values but with a different scale:

```
shuttleScale := moovScale DIV 10;
localDuration := GetTimeBaseStopTime(tBase, shuttleScale, tr);
localDuration :=
      localDuration - GetTimeBaseStartTime(tBase, shuttleScale, tr);
```

The shuttle control in TimeBaseSimple lets you scan the movie backward and forward. This is implemented by changing the current time value for the time base, which looks something like this:

```
SetTimeBaseValue(gTBState.tBase, value*10, gTBState.moovScale);
                                            (* 'movie scale/10' tick *)
```

Setting the rate. Although you can obtain the current rate for a time base and set the rate directly, for a time base associated with a movie a better approach is to make Movie Toolbox calls such as StartMovie or SetMovieRate. The Movie Toolbox executes these calls by changing the time base associated with the movie. For example, StartMovie gets the preferred rate and sets the time base rate to it, setting the movie's time base in motion.

When the movie is being controlled by the standard movie controller, it's important to call MCMovieChanged if you change any movie characteristic, such as the rate or the current time value, to keep the controller in sync with the new settings. As mentioned earlier, it's better to use high-level interfaces to enact these changes; for example, to change the rate via the movie controller, you can call MCDoAction(mc, mcActionPlay, newRate).

Using the time base flags. When you access a time base directly, you can set its movie to loop, either normally or backward and forward, by setting the time base flags. GetTimeBaseFlags retrieves the flags for inspection, and SetTimeBaseFlags modifies the flags. In TimeBaseSimple, the SetTBLoop routine sets the looping flags:

```
(* Changes the state of looping in the movie if needed. *)
PROCEDURE SetTBLoop(newFlags: LONGINT);
VAR      targetTB: TimeBase;
```

```
BEGIN
   targetTB := gTBState.tBase;           (* the movie's time base *)
   SetTimeBaseFlags(targetTB, newFlags);  (* change it *)
   gTBState.flags := newFlags;           (* remember new state *)
END;
```

Now that you've seen how you can access the state information of a time base, let's look at some of the possible uses of time bases.

TIME SLAVES

One interesting situation arises when you need to play back two or more instances of a movie simultaneously. In such situations you can synchronize the movies by enslaving all the instances to one time base. The central idea behind this is to have control of the movie's time flow pass through a single point instead of having a number of individual time bases running at the same time. The sample program TimeBaseSlave on the *Developer CD Series* disc shows how to do this.

TimeBaseSlave splits the window in which the selected movie is to play into four parts, with the quarters rotating while the movie is playing back. Figure 3 shows the TimeBaseSlave window at its four stages of playback.

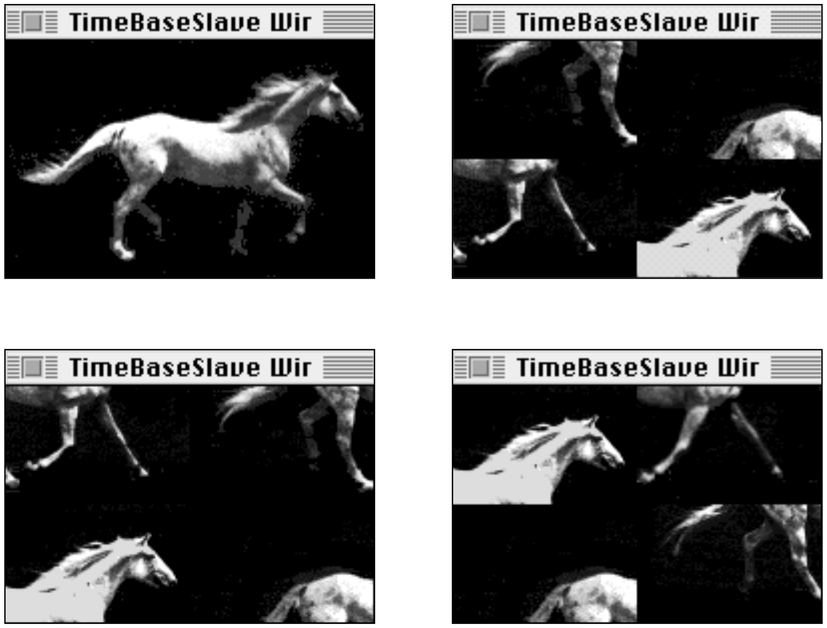


Figure 3
TimeBaseSlave Window at Its Four Stages of Playback

The basic programming strategy is as follows:

1. Get the time base associated with one of the instances of the movie.
2. Force the time base from step 1 to be used for the other instances.
3. Start playing the first instance of the movie, controlling it in any way you like. (TimeBaseSlave starts the movie and sets it back to the beginning when it reaches the end.)
4. The other instances of the movie will follow blindly.

The EnslaveMovies routine in TimeBaseSlave takes care of all this:

```
FUNCTION EnslaveMovies: OSErr;
VAR      err:               OSErr;
         masterTimeBase:    TimeBase;
         slaveZero:         TimeRecord;
         slaveZeroTV:       TimeValue;
         masterScale:       TimeScale;
         count:             INTEGER;
BEGIN
   err := noErr;
   masterTimeBase := GetMovieTimeBase(gMoov[1]);
                              {* time base of first movie instance *}
   masterScale := GetMovieTimeScale(gMoov[1]);
                              {* needed for SetMovieMasterTimeBase *}
   slaveZeroTV :=
      GetTimeBaseStartTime(masterTimeBase, masterScale, slaveZero);
                                                          {* ditto *}
   FOR count := 2 TO 4 DO      (* slave all movies to first time base *)
      BEGIN
         SetMovieMasterTimeBase(gMoov[count], masterTimeBase, slaveZero);
                                                    {* now we do it *}

         (* real programmers do check for errors *)
         err := GetMoviesError;
         IF err <> noErr THEN
            BEGIN
               ErrorControl('SetMovieMasterTimeBase failed');
            LEAVE;
            END;
      END;
   EnslaveMovies := err;
END;
```

Once the slave instances of the movie have been set to obey the first time base, their behavior will mimic the first movie's actions. In the TimeBaseSlave code, it appears that only the first instance is started and that only it is rewound when the end is reached. These actions are accomplished in TimeBaseSlave by calls to StartMovie and GoToBeginningOfMovie, respectively, with the first movie passed as a parameter.

You could use this technique to play different movies but have all of them under a single control. It might also be useful when no movies are involved at all but time bases are being used for timing purposes.

TIMELY TASKS

TimeBaseSlave also shows how to take advantage of the callback capabilities of time bases. Callbacks are useful when an application needs to execute given tasks when the time base passes through certain states. You can program time base callbacks to be triggered under the following conditions:

- when a certain time value is encountered (callBackAtTime)
- when a rate change occurs (callBackAtRate)
- when there's a jump in time (callBackAtTimeJump)
- when the start or stop time is reached (callBackAtExtremes)

Passing callBackAtTime to NewCallBack shows the use of callbacks that are executed at a specified time value. TimeBaseSlave uses the callback service to rotate the movie pieces at regular intervals; we ask to be called every three seconds in movie time.

Note that the time value triggering the callback depends on the rate of the time base. In other words, the time value specified will never be reached if the movie isn't playing (if the rate is 0). If the rate is something other than 1.0 (if the movie is accelerated or is moving in slow motion or in reverse), the specified break will come every three seconds in movie time, not clock time.

CREATING A CALLBACK

First TimeBaseSlave has to create a callback. This could be accomplished as follows:

```
cb := NewCallBack(tb, callBackAtTime);
```

Since we want to be called at interrupt time, however, the line looks like this:

```
cb := NewCallBack(tb, callBackAtTime + callBackAtInterrupt);
```

The variable cb receives a callback, which depends on the time base tb. The callback will be executed at specific times and can be scheduled to fire at interrupt time.

NewCallBack moves memory, which means that you can't create a callback while in an interrupt handler. Electing to be called at interrupt time has an advantage over normal interrupt-driven tasks, however, as I'll explain later.

PRIMING THE CALLBACK

Once we're satisfied that the callback was created (cb <> NIL), we proceed to prime the callback. At this point we have only the hook into the time base; priming the callback schedules it to call us. This is accomplished by CallMeWhen, as follows:

```
err := CallMeWhen(cb, @FlipPieces, callWhen, triggerTimeFwd, callWhen,
         scale);
```

FlipPieces is the routine that we want to have called when the specified time value arrives. The callWhen variable is passed both as a refCon (the third parameter) and as the time to trigger the callback (the fifth parameter), the idea being that FlipPieces will need to know the current time. Of course, the refCon parameter can also be used for any other purpose you may see fit.

The time at which the callback is triggered is given a frame of reference by the scale parameter. Remember that a time value without a time scale has no meaning at all. Finally, triggerTimeFwd means that our routine will be called only when the movie is moving forward. This is reasonable since TimeBaseSlave plays back the selected movie in forward motion only.

THE FLIPPIECES ROUTINE

The routine responsible for servicing the callback follows a simple interface and is defined in TimeBaseSlave as follows:

```
PROCEDURE FlipPieces(cb: QTCallBack; refCon: LONGINT); (* CallBackProc *)
(* The refCon parameter contains the time that triggers the callback;
this is the value passed to the CallMeWhen routine. *)
VAR   j:          INTEGER;
      callWhen:   LONGINT;
      scale:      TimeScale;
      stop:       LONGINT;
      tr:         TimeRecord;
      tb:         TimeBase;
      err:        OSErr;
BEGIN
   stage := (stage + 1) MOD 4;
   FOR j := 1 TO MoviePieces DO
      ShiftMoviePieces(j); (* turn the movie pieces around *)
   scale := 100;           (* 100 units in this scale means 1 second *)
   callWhen := refCon + 3*scale;    (* call me in 3 seconds *)
   tb := GetCallBackTimeBase(cb);   (* needed for next line *)
   stop := GetTimeBaseStopTime(tb, scale, tr);
   IF callWhen > stop THEN    (* wrap around the three seconds *)
      callWhen := GetTimeBaseStartTime(tb, scale, tr) + callWhen - stop;

   (* now to really reprime the callback *)
   err := CallMeWhen(cb, @FlipPieces, callWhen,
      triggerTimeFwd + callBackAtInterrupt, callWhen, scale);
END;
```

TimeBaseSlave does the actual splitting of the movie into different views by creating four instances of the same movie and setting the movie clipping region for each one to be the rectangle in which each is expected to display. When it's time to move the pieces, the movie box of each instance is offset to cover the next spot. Take a look at SplitMovie and ShiftMoviePieces to see the code.

A FEW CONSIDERATIONS

Inquisitive readers will have noted that when calling CallMeWhen, TimeBaseSlave uses both noninterrupt and interrupt-time invocations. This was done to illustrate one of the advantages of using Movie Toolbox callbacks: the Toolbox takes care of setting up the A5 world when your service routine is called. Having the A5 world set up properly is useful when your program needs to access global variables; other interrupt handlers can't count on A5 being right when they're invoked.

Using time base interrupt callback routines does not, however, liberate the application from the normal limitations of interrupt-servicing routines; for example, you can't move memory.

As mentioned earlier, although time bases are created automatically when a movie is opened or created, they can also exist on their own. If an application requires services that allow control over the passing of time, it can create a time base and use callbacks to trigger the service routines required. Keep in mind that even when a time base has no movie, the application must still call MoviesTask to guarantee that callback routines will get time to run.

OTHER TYPES OF CALLBACKS

Time base callbacks can also be triggered by a change in the rate or by a jump in the time value. A change in the rate occurs when the movie is stopped while it's playing, when a movie is set in motion, or when the playback speed is somehow changed. A jump in time occurs when the current time value in the time base is set to a value outside the normal flow — for example, when a movie that's playing is set back to the beginning. In addition, QuickTime 1.5 introduces callbacks "at extremes" that can be triggered when the time base time reaches the start or stop time.

These three means of triggering a callback are of interest only if the code is tracking the behavior of the movie, as a movie controller or a media handler would need to do;

the constants used for calling NewCallBack in these cases are callBackAtRate, callBackAtTimeJump, and callBackAtExtremes.

FINALLY

If you'd like to play with the sample programs, you may want to try some variations. For instance, it's very easy to modify TimeBaseSlave to have all the movies play at their own beat, with separate time bases, and compare the performance with the original TimeBaseSlave. You could also modify TimeBaseSimple to see the time values obtained with different time scales.

Time bases are an important part of the QuickTime Movie Toolbox. Understanding their role in the way movies play back can be extremely important for developers trying to push the envelope in writing new and innovative QuickTime applications. This article has opened the door; now it's up to you to decide whether this route will prove beneficial to your efforts.

HIGHLIGHTS OF QUICKTIME 1.5 NEW FEATURES

Listed below are some of the more significant features of QuickTime 1.5.

- Photo CD: Using QuickTime 1.5 and the Photo CD Access extension, you can work with Kodak Photo CDs on your Macintosh. Photos on the CD appear as standard PICT files and can be opened in any application that opens pictures.
- Compact video compressor: A new compressor has been added that provides high-quailty, low data rate playback.
- Movie import: Any application that opens movies using QuickTime's Standard File Preview can import PICS, AIFF, PICT, and System 7 sound files.
- 1-bit dither: Playback performace of color movies has been significantly enhanced on black-and-white (1-bit) displays. This is particularly useful on PowerBook computers.
- Sequence grabber dialogs: To provide for a more flexible and consistent user interface for configuring capture devices, the sequence grabber provides a set of standard configuration dialogs. Support for sound capture is also substantially improved.
- Text media handler: In addition to sound and video, QuickTime 1.5 has built-in support for text. The text media handler is built using QuickTime's new Generic Media handler, which allows you to create your own QuickTime data types.
- Standard compression: The Standard Image Compression dialog is now built into QuickTime. The user can pan and zoom the test image within Standard Compression.

THANKS TO OUR TECHNICAL REVIEWERS
Bill Guschwan, Peter Hoddie, David Van Brink•

GRAPHICAL TRUFFLES

ANIMATION AT A GLANCE

EDGAR LEE

The Macintosh has always provided animation capabilities. From the early Macintosh 128K to current CPUs, animation has consistently played a large part in the development of software. And though CPU models continue to change, the theories and concepts behind animation have stayed basically the same. Simply stated, animation is the process of stepping through a sequence of images, each slightly different from the previous.

The thought of animation on the Macintosh usually brings to mind games and multimedia, when in fact the actual use of animation is more prevalent than most people imagine. I'll describe some common uses and methods of performing animation and get you started on writing your own animation sequences.

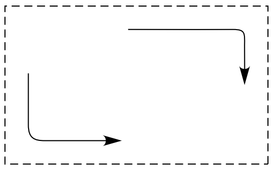
METHOD 1: PRIMITIVE BUT EFFECTIVE

One of the most fundamental methods of animation is using the srcXor transfer mode. The basic idea is that once you've drawn something in this mode, you can erase it simply by drawing it again, restoring the bits underneath to their previous state. Primitive though it may be, this method is common to many applications. Probably the most obvious example of it can be found in the Finder. Familiar to even the novice Macintosh user is the dotted rectangle that often appears during desktop navigation. The movement of the dotted rectangle, which appears when the user selects multiple icons or drags windows across the desktop, is a simple form of animation. The dotted rectangle is also used to create the zooming effect when desktop folders are opened and closed.

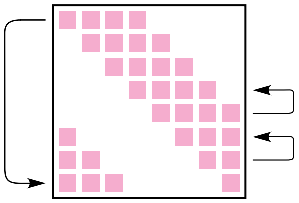
To use this method, you set the current transfer mode to srcXor before drawing the object you plan to animate. In the desktop example, the Finder switches to srcXor mode and then draws the dotted rectangle with a simple FrameRect call, with the PenPat set to 50% gray. The movement of the dotted rectangle is accomplished by redrawing the rectangle at its previous position before drawing it at its new location. With srcXor mode, simply redrawing the rectangle at the same position restores the desktop to its original state. So by repeatedly drawing and redrawing the rectangle in its new position, you float a frame across the screen without damaging the contents of the desktop.

As a variation on the dotted rectangle, applications use what's called the "marching ants" effect. With this effect, the bounding frame gives the illusion that the dashed lines or "ants" are moving around the edges of the box, thereby producing an animated and more interesting visual appearance.

The marching ants effect is simple to create. The most common way to do this is with a simple 8-by-8-bit pattern. To create the illusion, you draw a bounding frame by calling FrameRect, with the PenMode set to srcXor and the PenPat set to a pattern defined with diagonal stripes (see the illustration below). Shifting the pattern up one row, and then wrapping the first row of the pattern to the last row, creates the effect. If the rows were shifted down rather than up, the ants



Marching ants



8-by-8-bit pattern

EDGAR LEE (AppleLink EDGAR) Recently spared from the traumas of big city living, Edgar enjoys the relaxing and granola-like atmosphere of sunny Cupertino. When asked what he likes most about the area, he proudly points to his car stereo in disbelief that it's still there. Besides adjusting to his newly found appreciation of suburban living, Edgar enjoys a good challenge of doubles volleyball, an excellent head-to-head game of Tetris, and learning the newest and latest human tricks from his faithful companion, Sunny. Though Edgar realizes Sunny is only a dog, he still believes some of the engineers here at Apple could stand to learn a lot from her. Of course these engineers don't seem to agree.•

would appear to move in the opposite direction. In either case, the ants typically start at one corner of the box and then end at the opposite corner.

As with the dotted rectangle, the frame is continually drawn and redrawn, but this time with each new updated pattern. Note the difference between the two effects when the frame is drawn: With the ants, the frame is constantly being drawn and redrawn even if the rectangle's coordinates haven't changed. With the dotted rectangle, the frame is redrawn only when its position has changed. Since no animation takes place when the dotted rectangle is sitting in the same position, it's not necessary to continually draw the frame in that case.

METHOD 2: NOT SEEING IS MORE THAN BELIEVING

Another method of performing animation is to use off-screen drawing. With this method, the actual drawing is being done behind the user's back. The animation frames are prepared off-screen and quickly transferred to the screen with CopyBits to create the animation sequence. Regardless of what CPU you're running, this method can provide excellent animation effects. And with the advent of GWorlds to simplify the process of building off-screen environments, performing animation with this technique has become much easier.

In this section I'll provide some important points to consider when building your own off-screen world and describe how to apply these off-screen worlds to animation. For a detailed description of creating your own custom GDevices, cGrafPorts, and pixMaps, see the Macintosh Technical Note "Principia Off-Screen Graphics Environments."

Before even considering off-screen animation, you need to determine whether your Macintosh has enough memory for creating the off-screen environment. Without sufficient memory, you might as well forget it. Having high-performance, high-quality animation isn't cheap. Most of what determines the amount of required memory is the off-screen world's dimensions and pixel depth.

- Typically, or at least for this method, the dimensions of the off-screen world are the same as those of the entire on-screen area.
- For the depth of the off-screen world, you'll need to determine whether it's based on the depth of the images used in the window or on the depth of the GDevice intersecting the window. In the case where the GDevice is set to direct colors, you may want to create only an 8-bit off-screen world to save memory if your images use only 256 or fewer colors. On the other hand, you may want to create an off-screen world equal to the depth of the GDevice containing the window, for better data transfer performance.

Once you've determined the dimensions and depth for the off-screen world, you're ready to create the off-screen environment. Note that if you're using the GWorlds introduced with 32-Bit QuickDraw, many of the off-screen initialization procedures have been simplified. Also, with certain video display cards, the GWorlds can be cached into the NuBus™ card's memory, providing even better performance when off-screen worlds are used.

To create the off-screen environment, you pass NewGWorld the off-screen dimensions, depth, and color table, and the routine creates the environment or warns you if there wasn't sufficient memory. After you've made all the required memory checks and created your off-screen environment, either by hand or with NewGWorld, the next step is to create the animation sequence.

In the simplest case, the off-screen world is used to store an identical copy of what's displayed on the screen. Rather than erasing and drawing the moving object on-screen, you perform all this in the off-screen world. Once the moving object has been drawn in its new position, the off-screen image is transferred to the screen. By continually drawing the next frame of the moving object in the off-screen world before displaying

For more information on caching GWorlds into NuBus memory and improving drawing performance, see "Macintosh Display Card 8•24 GC: The Naked Truth" in *develop* Issue 3.•

it on the screen, you produce the animation effect. The following steps describe the process.

1. Assuming that the entire window is being used for the animation, create an off-screen environment of the same dimensions as the window, either by hand or with NewGWorld. When you're defining the depth and color table of the off-screen world, remember that QuickDraw requires extra time to map colors when the destination GDevice's depth and color table are different from those of the source.
2. Switch to the off-screen grafPort and GDevice and draw the background image. This is the image that the object will be moved on top of; typically it won't change.
3. Draw the object that will be moved or animated into the off-screen world. Actually, any image not part of the background image should be drawn at this time. Also, since the object overwrites the background image, the background under the object will eventually need to be restored.
4. Switch back to the on-screen grafPort and GDevice and use CopyBits to transfer the off-screen pixMap to the screen.

These steps create just one frame of the animation sequence. To create the full sequence, repeat the last three steps until the animation is complete. In step 2, instead of redrawing the entire background, you may want to redraw just the areas that need to be restored, if that information is available. By redrawing just a portion of the damaged background, you'll notice improved performance, especially when working with higher pixel depths.

Besides providing a quick introduction to off-screen animation, this method has the advantage that it's simple and straightforward. Since all the objects and images are drawn at one time and in the same environment, it's easy to create your sequences and synchronize the animation for any moving object. However, as mentioned earlier, large off-screen images at higher pixel depths can really affect the performance of the animation. To overcome this problem, you need to use multiple off-screen worlds.

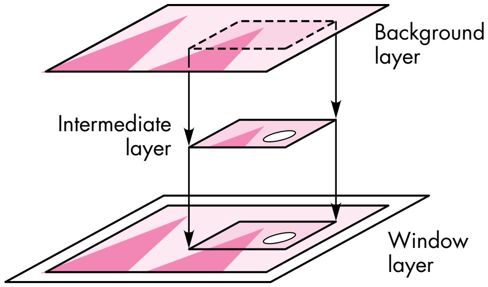
METHOD 3: SWITCHING INTO HIGH GEAR

The concept of multilayer off-screen worlds isn't much different from the basics of off-screen animation. Rather than having just one off-screen environment, you've also got an intermediate off-screen layer in which all the actual drawing is completed, leaving the background layer undamaged. So unlike the previous method, where one off-screen world was used for storing the background and the moving object, this method uses two separate off-screen worlds to maintain this information. The following steps describe how the intermediate layer fits in.

1. Again, create the background off-screen layer with the same dimensions as the window.
2. Switch the current grafPort and GDevice to the background layer, then draw the background image. This layer will never change, since its main purpose is to restore the overwritten areas of the intermediate layer.
3. Find the common rectangle containing the object's previous location and its new location. This can be calculated by passing UnionRect the object's bounding rectangle for both positions. Be sure the common rectangle uses coordinates local to the window.
4. Create the intermediate off-screen layer with the dimensions of the common rectangle.
5. Switch to the intermediate layer and transfer the area of the corresponding common rectangle of the background layer to the current layer. This will restore the area at which the object was last positioned. Rather than having to redraw the background for each frame, you simply replace the damaged area with the background image stored in memory.
6. Draw the moving object at its new location in the intermediate layer. If multiple objects are within the same bounding region of this layer, they should be drawn at this time as well.

7. Switch to the window layer and use CopyBits to transfer the contents of the intermediate layer to the screen.

Finally, to create the entire animation sequence, repeat steps 3-7 until the animation is complete. The illustration below shows the process of creating one of the frames in the sequence. In this frame, the moving object is the sun, drawn on top of the background image of the mountains.



When moving multiple objects, you'll need to decide whether to handle the objects separately or in groups. In the case where objects are widely dispersed in the window, it would be more practical to create a separate intermediate layer for each object than to create one layer containing all the objects. Since no changes are occurring in places between widely spread objects, unnecessary time and memory would be spent updating these areas.

However, if the objects are closely spaced, grouping the objects and creating one intermediate layer would make more sense. Since objects can overlap each other, creating separate off-screen worlds would not be too practical or easily accomplished. So when determining the number of intermediate off-screen layers, you'll want to first check where the objects are located in the window.

The main advantage of using the intermediate layer is the performance improvement. As mentioned earlier, transferring large blocks of data at high pixel depths can be time consuming. As you can guess, the smaller the transfer image, the less time QuickDraw requires. Another advantage of using this layer is the ability to isolate the background image. Since all the drawing is taking place in the intermediate layer, there's no need to redraw the background image for each frame, which can be a real time saver for complex backgrounds. Though more memory is required with the addition of the intermediate layer, the performance gains can sometimes make the extra memory worth it.

Finally, to fully optimize the animation performance, you'll want to be sure the data transfer from the off-screen layers is as fast as possible. Since you can influence the speed of CopyBits, here are a few points you'll want to keep in mind when creating and using off-screen layers:

- For indexed GDevices, the same color table should be used for the window and the off-screen layers. Since no color mapping should be required when the source and destination share the same color table, less time is needed for the data transfer.
- Be sure no nonrectangular clipping is involved in the CopyBits operation. Having to check which pixels should or shouldn't be clipped can really slow down the data transfer.
- Use srcCopy as the transfer mode for CopyBits. Any other mode takes extra time to perform the logical operations on the source and destination pixels.
- Set the current port's foreground color to black and background color to white before calling CopyBits. This will ensure that no colorizing (which can be slow) takes place.
- Make sure no dithering takes place. Unless you have your own rippin' fast method for dithering, try to stay away from it. If possible, prepare the images in the off-screen layers in such a way that dithering isn't needed.
- Keep the same alignment of pixels for the source and destination pixMaps. Having to shift unaligned pixels can take time.
- The source and destination rectangles should be the same size. Scaling requires extra work.

For source-code routines that create and manage off-screen layers, see GWLayers in the Sample Code folder on the *Developer CD Series* disc. To see how these routines are actually used, check out the Kibitz and DTS.Draw samples on the CD as well. (GWLayers is brought to you by Forrest Tanaka, and Kibitz and DTS.Draw are from Eric Soldan.)•

By following as many of these points as possible, you'll improve the performance that you'll get out of CopyBits and waste less time in the on-screen updates.

LIGHTS, CAMERA, ACTION!

I've presented several methods of animation; which method to use depends on your application. In fact, you may choose to use several methods or switch between methods under different system requirements. Say your application uses multiple layering for optimal animation; under low-memory conditions, you may want to switch to just one off-screen world to provide at least some type of off-screen animation. But if that isn't even an option, you may have to do all the animation on-screen. For an example that does exactly that, see DTS.Draw in the Sample Code folder on the *Developer CD Series* disc. If sufficient memory is available to create the off-screen worlds, the application uses the multilayer method; otherwise, the application decides on the next best method based on the current available memory.

This column has described different animation techniques, but the principle behind them is basically the same, even if the results don't show it. Given a set of slightly different images, all the methods involve stepping through the series of images, where each object in the image is erased before the next object in the series is displayed.

Animation provides excellent visual effects, more fun for the programmer, and most important, an enhanced experience for the user. Now that you've got the basics of animation on the Macintosh, I hope you'll be inspired to animate your own applications!

RECOMMENDED READING

- "Macintosh Display Card 8•24 GC: The Naked Truth" by Guillermo Ortiz, *develop* Issue 3.
- Macintosh Technical Notes "Principia Off-Screen Graphics Environments" (formerly #120) and "Of Time and Space and _CopyBits" (formerly #277).
- *Computer Graphics: Principles and Practice,* 2nd ed., by J. D. Foley, A. Van Dam, S. K. Feiner, and J. F. Hughes (Addison-Wesley, 1990), Chapter 21.

For a more detailed explanation of increasing data transfer performance, see the Macintosh Technical Note "Of Time and Space and _CopyBits."•

Thanks to Bill Guschwan, C. K. Haun, Guillermo Ortiz, Konstantin Othmer, Forrest Tanaka, and John Wang for reviewing this column, and to Brigham Stevens for his special help.•

BETTER APPLE EVENT CODING THROUGH OBJECTS

In "Apple Event Objects and You" in develop *Issue 10, Richard Clark discusses a procedural approach to programming for Apple events and goes into details of the Apple event object model. This article reveals a few simple truths about the significance of Apple events and the Apple event object model, focusing on how the object model maps onto a typical object-oriented application. It also provides an object-oriented C++ framework for adding scripting support.*

ERIC M. BERDAHL

It's every developer's worst nightmare: Your team has just spent the last two years putting the finishing touches on the latest version of Turbo WhizzyWorks II NT Pro, which does everything, including make coffee. As a reward for your great work, the team is now preparing to do some serious tanning development on an exotic island. Then, Marketing comes in with "one last request." They promise it's the last thing they'll ask for before shipping, and in a weak moment, you agree that one last little feature won't hurt your itinerary. "Good," quips the product manager, "then as soon as you add full scripting support, you can enjoy your vacation."

You know that to add scripting support, you need to delve into Apple events. You think this requires learning about Apple events, the Apple event object model, and scripting systems. Further, you think Apple events must be designed into your application from the ground up and can't possibly be added without a complete redesign. Which of the following is the appropriate reaction to Marketing's request?

A. Immediately strangle your sales manager and plead justifiable homicide.

B. Look around while laughing hysterically and try to find the hidden Candid Camera.

C. Change jobs.

D. Feign deafness.

E. None of the above.

ERIC M. BERDAHL (AppleLink BERDAHL) is a refugee from Chicago, recently deported to the West Coast to join Taligent. Having lived most of his life in a suburb of the Windy City, he exhibits a psychosis common to that area of the country — fanatic loyalty to the Cubs. His formula for success includes bucking the establishment and blindly following one's heart over one's head. The jury's still out on whether that formula works, but it's been effective so far. He's the current president of MADA, an international developer's association devoted to providing cutting-edge access to information about object technologies. MADA conferences are a real blast, too (just ask Eric about his grass skirt). In his copious spare time, he collects comic books, catches up on the Cubs' latest follies, and chases a neurotic flying disc around a grassy field (some call it Ultimate).•

Unfortunately, there's no correct answer, but the scenario is all too real as developers are increasingly being asked to add scripting support to their applications. The design of Apple events and the Apple event object model can provide the user with more power than any other scripting system. However, to access the power of the design you need to work with the complex interface provided by the Apple Event Manager. By its nature, this interface collapses to a procedural plane of programming that prevents developers from fully taking advantage of the object-oriented design inherent in the Apple event world. The Apple event object model is difficult to implement without some fancy footwork on the part of your framework. But remember the words of Marshall Brodeen, "All magic tricks are easy, once you know the secret." With this in mind, join me on a trip through the rabbit hole into AppleEventLand.

WHAT ARE APPLE EVENTS AND THE OBJECT MODEL?

Whenever I give presentations on Apple events, the audience has an overwhelming urge to ignore the theory and jump into coding. Resist the urge. For most developers Apple events provide an unfamiliar perspective on application design. To appreciate the significance of Apple events and the object model, it's important to understand their underlying concepts and background. So, although you'll be reading about code later, a little theory needs to come first.

At the most basic level, Apple events are a program-to-program communication (PPC) system, where *program* is defined as a piece of code that the Macintosh can see as an application (in other words, that has a real WaitNextEvent-based event loop). However, billing Apple events as PPC is akin to describing an F-16 as merely a plane. To fully understand how Apple events are more than simple program-to-program communication, you need to take a look at the Apple event object model.

The object model isn't really defined in a pithy paragraph of *Inside Macintosh*, but is instead a holistic approach to dealing with things that users call objects. In a literal sense, the object model is a software developer's description of user-centric objects or *cognitive objects*.

COGNITIVE THEORY

Cognitive science tells us that people interact with the world through objects. A printed copy of *develop* is an object, a plant in the corner of your office is an object, and a can of Coke Classic on your desk is an object. Each of the objects has properties, behaviors, and parts. Some properties exist for each of the objects (for example, each one has a *name*) and other properties make sense for only some of the objects (for example, *page size* makes sense only when applied to *develop*). Behaviors are quite similar to properties in their ephemeral binding to objects. Only Coke will *fizz*, but all three objects will *decompose*. However, they each *decompose* in a different way. Further, each object can be separated into arbitrary parts that are themselves objects. The plant can be separated into branches, which can in turn be separated

Marshall Brodeen, a.k.a. Wizzo the Wacky Wizard from station WGN's "Bozo's Circus," was a television spokesman for T.V. Magic Cards.•

into leaves. The plant itself can also be separated into leaves, so leaves are contained by both branch objects and plant objects.

BACK INSIDE THE COMPUTER

Now, since a user will someday interact with your software, and since users interact with the world in terms of cognitive objects, it makes sense to model software in terms of cognitive objects. Hence, the object model describes objects in a rather ghostlike fashion whereby objects have behaviors and properties and contain other objects. Although the object model defines an inheritance for each category of objects (for example, Journal might inherit from OpenableThing which might inherit from Object), it's used only for the purpose of grouping similar behaviors. Just as in the mind, the only thing that's important is the identity of a specific object in existence at a given time — its categorization is purely a detail of implementation.

Gee, this sounds a lot like what *real* programmers mean when they talk about objects. Strangely enough, real objects and cognitive objects are quite related. Many references cite cognitive theory as justification for beginning to program in an object-oriented style. Object-oriented code tries to get closer to the language of the native operating system of the human mind than traditional procedural approaches, and the format of an Apple event object mirrors natural language to a surprisingly large degree. It comes as no surprise, then, that Good Object Design lends itself quite easily to slipping in support for Apple event scripting.

APPLE EVENT OBJECTS AND SCRIPTING

The motivation for you to provide object model support is so that your users can "script" your application. There are a variety of solutions available today that allow advanced users to write things that resemble DOS batch files or UNIX® shell scripts. These entities are commonly called *scripts*, but in the context of Apple events a script is something with greater potential. Whenever a user thinks "I want to sharpen the area around the rose in this picture," a script has been formed. If this seems too simplistic, consider it again. *Script* here refers to the earliest conception of a user's intent to do something. It's not relegated to the world of the computer and does not imply any given form or class of forms; an oral representation (voice interface a la the Knowledge Navigator) is equally as valid as a written one (traditional scripting systems). From this perspective, the definition of *script* takes the user to a greater depth of control over applications than previously dreamed of, allowing access to the very engine of your application by the very engine of the user. This is the great empowering ability of Apple events: they enable users to use their native operating system — the mind — with little or no translation into computerese.

OBJECT-ORIENTED PROGRAMMING OBJECTS

The biggest problem with Apple event objects is the interface provided by the Apple Event Manager. Instead of allowing you to write real object-oriented source code

Good Object Design is sometimes lumped together with pornography as being difficult to define, "but I'll know it when I see it." Others consider the search for G.O.D. as a holy crusade. Rather than giving a thoroughly useless description for G.O.D. here, I refer the interested reader to *Developing Object-Oriented Software for the Macintosh* by Alger and Goldstein (Addison-Wesley, 1992).•

using a given class library that implements basic Apple event and object model functionality, the Apple Event Manager requires you to register every detail programmatically. You must declare what classes exist, which methods exist and where, and what relationships are possible within and between classes. Although at first this flexibility seems advantageous, many developers find it a problem later when they have to declare everything again at run time. Anyone with secret desires to design an object-oriented runtime environment and a compiler/linker combination to support that environment will feel quite at home with Apple event coding.

The second biggest problem with Apple event objects is that programs aren't written in the Apple event (user) world. Instead, they're often written in object-oriented programming languages like LISP and C++. What's needed is a good generic interface to translate objects from the user world of natural language into the world of LISP or C++ objects. Scripting systems do some of the work by delivering Apple event objects to applications in the form of object specifiers, a strange data structure that resembles a binary form of natural language stuffed into the familiar Apple event generic data structure AEDesc. However, object-oriented applications ship objects around in the form of . . . well . . . objects! So, you need translation from binary natural language to actual objects. Easy, huh? (Don't hurt me yet — this will seem fairly straightforward after reading a bit further.)

Presenting a new interface should solve the problem of the Apple Event Manager interfaces. Presenting that new interface in terms of the familiar object-oriented class libraries should solve the problem of different paradigms. So, if these two problems are approached with an object perspective, it's clear that some of the classes in your program need to include a set of methods that implement object model protocols. Application domain classes must be able to return objects contained within them and to perform generic operations on themselves. It turns out that if your classes also provide the ability to count the number of a specific type of object they contain, you can provide a rudimentary, yet powerful, parsing engine for transforming objects from the Apple event world into the traditional object programming world.

Further analysis indicates that only those application domain classes that correspond to object model classes need this protocol. This indicates that the protocol for providing Apple event object model support is probably appropriate to provide in a mixin class (a class that's meant to be multiply inherited from). In this way, only those classes that need to provide object model support must provide the necessary methods. In the sample application discussed later, that class is called MAppleObject. MAppleObject plays a key role in UAppleObject, a generic unit that can be used to provide Apple event object model support to any well-designed C++ application.

Apple provides a convenient solution to the user versus programming language problem in the form of the Object Support Library (OSL). The OSL has the specific responsibility of turning an object specifier into an application's internal representation of an object. (See "A Sample OSL Resolution" for an example of how

AEDesc is the basic Apple event data structure described in *Inside Macintosh* Volume VI, Chapter 6, "The Apple Event Manager."•

A SAMPLE OSL RESOLUTION

Here's a short example to give you a feel for how the OSL actually works. Don't read too much into the details of object resolution, but do try to understand the flow and methodology the OSL applies to resolve object specifiers. Also, don't worry too much about how the OSL asks questions; the protocol you'll actually be using in UAppleObject hides such details from you.

Figure 1 on the next page gives an overview of the process. Consider the simple object specifier "the third pixel in the first scan line of the image called 'Girl with Hat,'" and an Apple event that says "Lighten the third pixel in the first scan line of the image called 'Girl with Hat' by twenty gray levels." On receiving this Apple event (Lighten) the application notes that the direct object of the event (the third pixel in the first scan line of the image called "Girl with Hat") is an object specifier and asks the OSL to resolve it into a real object.

At this point the parsing engine in the OSL takes over, beginning a dialog with your application through a set of preregistered callback routines. Notice that the object specifier bears a striking resemblance to a clause of natural language — English in this case. This is not unintentional. Apple event objects are cognitive objects, and cognitive objects are described by natural language — hence the parallels between object specifier formats and natural language. Further, the parsing engine inside the OSL operates like a high school sophomore parsing sentences at the chalkboard. But I digress . . .

To continue, the OSL asks the null object to give it a token for the image called "Girl with Hat." (Tokens are the Coin of the Realm to the OSL.) So the null object looks through its images to find the one named "Girl with Hat" and returns a token to it.

The OSL then turns around and asks the image called "Girl with Hat" to give it a token for the first scan line. After getting this token, the OSL has no further use for the image token, so it's returned to the application for disposal. In effect, this says, "Uh, hey guys, I'm done with this token. If you want to do anything like free memory or something, you can do it now." Notice how polite the OSL is.

Next, the OSL asks the scan line for a token representing the third pixel, which the line handily returns. Now it's the scan line token's turn to be returned to the application for recycling. The OSL has no further use for the scan line token, so the application can get rid of it if necessary.

Finally, having retrieved the token for the third pixel of the first line of the image called "Girl with Hat," the OSL returns the token with a "Thanks, and come again." The application can then ask the object represented by the token to lighten itself (remember that was the original Apple event), and dispose of the token for the pixel.

As you can see, the OSL operates by taking an unreasonable request, "give me the third pixel of the first line of the image called "Girl with Hat," and breaks it into a number of perfectly reasonable requests. Thus, your application gets to take advantage of its innate knowledge of its objects and their simple relationships to answer questions about complex object relationships.

the OSL actually works.) The OSL implements a generic parsing engine, applying a few simple assumptions about the state of the application's design to the problem. However, for all the power provided by the engine within the OSL, it lacks an object-oriented interface. Instead, it uses a paradigm like that provided by the Apple Event Manager, requiring the application to register a set of bottleneck routines to provide application-specific functionality. As with the Apple Event Manager, you must write

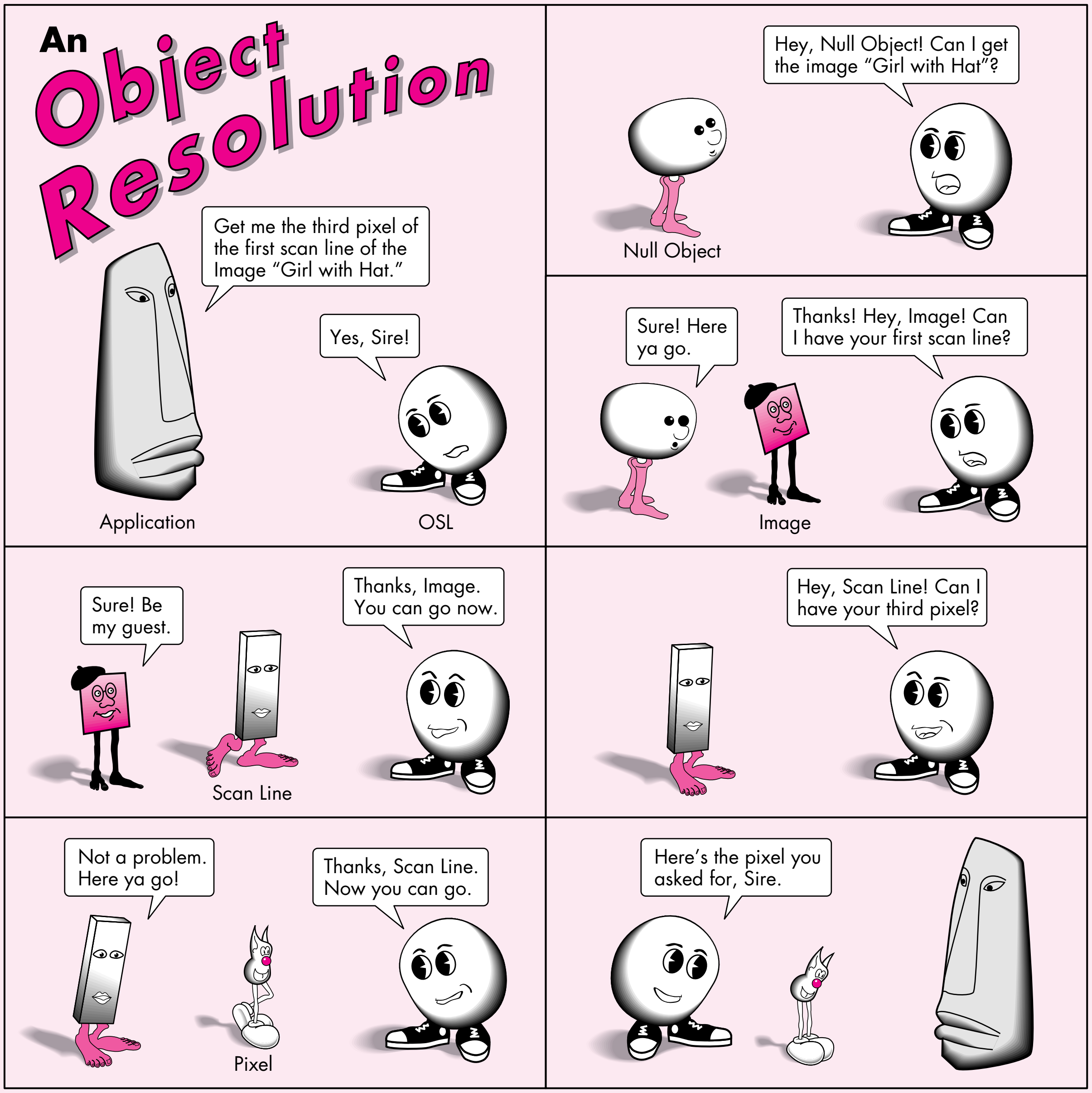

An Object Resolution
Get me the third pixel of the first scan line of the Image "Girl with Hat."
Yes, Sire!
Application
OSL
Hey, Null Object! Can I get the image "Girl with Hat"?
Null Object
Sure! Here ya go.
Thanks! Hey, Image! Can I have your first scan line?
Image
Sure! Be my guest.
Thanks, Image. You can go now.
Scan Line
Hey, Scan Line! Can I have your third pixel?
Not a problem. Here ya go!
Thanks, Scan Line. Now you can go.
Pixel
Here's the pixel you asked for, Sire.

Figure 1
Resolving an Object Specifier

routines that implement runtime dispatching to the individual objects your application creates instead of using the natural method-dispatching mechanisms found in your favorite object-oriented language, whatever it may be.

The nicest thing about the OSL is that, like the Apple Event Manager itself, it applies itself quite well to being wrapped with a real object-oriented interface (although you have to write it yourself, sigh). Curiously, the OSL solves both problems — poor interface and cognitive versus object-oriented programming differences. With a nice object-oriented framework, you can write your code once, in the fashion to which you're accustomed. I won't lie to you by telling you the job becomes easy, but it does change from obscure and harrowing to straightforward and tedious.

OBJECT MODEL CONCEPTS

There are two basic concepts defined in the object model. One is *containment*, which means that every object can be retrieved from within some other object. In the language of the object model, every object is *contained by* another object. The only exception to this rule is the single object called the *null object*. The null object is commonly called the *application object*, and may or may not be contained by another object. In practice, a null object specifier is like a global variable defined by the object model. The application implicitly knows which object is meant by "null object." Object resolution always begins by making some query of the null object.

For example, with a simple image processor, it would be appropriate to state that pixels are contained by scan lines, scan lines by images, and images by windows. It's also appropriate to have pixels contained by images and windows. Windows themselves have no natural container, however. Therefore, they must be contained by the null object. One way you can decide whether these relationships make sense for your product is to ask if a user could find it useful to do something to "the eighth pixel of the second scan line" or to "the twentieth pixel of the image." If statements like these make sense, a containment relationship exists.

The second basic concept of the object model is *behavior*. Behavior is quite simple; it means that objects must be able to respond to an Apple event. Behavior correlates directly with the traditional object programming concept of methods of a class. In fact, as you'll see, the actual Apple event–handling method of Apple event objects is usually a switch statement that turns an Apple event into a dispatch to the C++ method that implements the Apple event's functionality.

Taken together, the concepts of containment and behavior define the limits for objects in the model of the Apple event world. The object model resembles the programming worlds of Smalltalk or LISP, where everything is an object. Everything. For those familiar with these paradigms where even integers, characters, and floating-point numbers are full first-citizen objects, the Apple event world will be a refreshing change from traditional programming in C++ and Pascal.

FINDING THE OBJECTS

The overriding concept in designing object model support in your application is to do what makes sense for both you — as the developer — and the user.

1. It's best to begin by deciding what objects exist in your application. To decide what objects exist, do some user testing and ask the users what objects they see and what objects they think of while using your application. If this isn't possible, just pretend you're a user and actually use your application, asking yourself those same questions. For example, if you ask users for a list of objects in an image processing application (and refrain from biasing them with computer mumbo jumbo) they'll probably list such things as window, icon, image, pixel, area, scan line, color, resolution, and menu bar. (Figure 2 shows types of objects a user might list.) Guess what? In reality, those probably are object model classes that an image processing application could support when it supports the object model. Since the objects you'll want to support are user-level kinds of entities, this makes perfect sense.

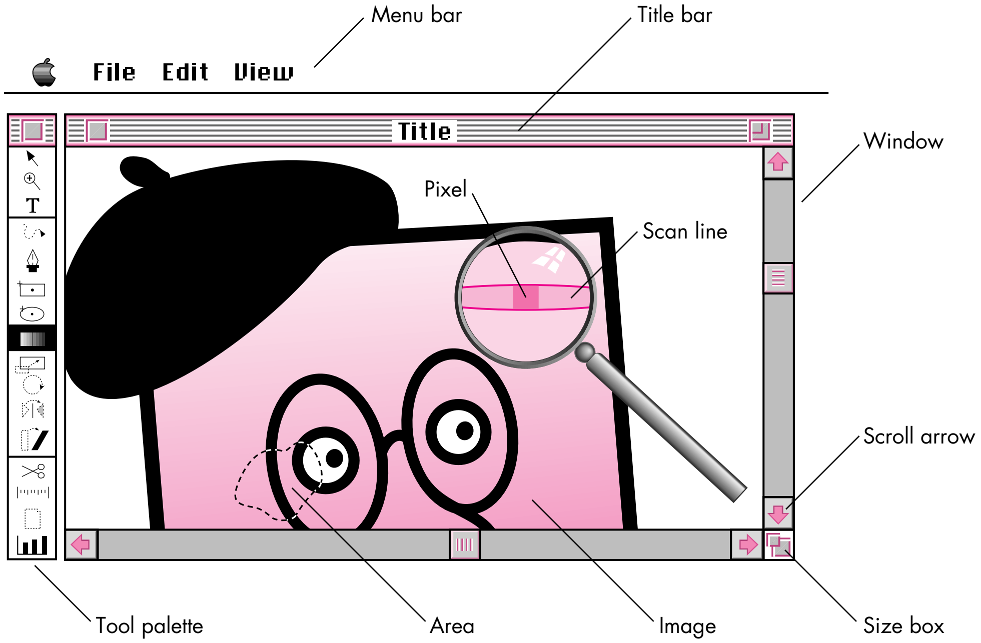


Figure 2
Objects the User Sees

2. After deciding what objects exist in your application, run another series of user tests to determine the relationships between different objects. For example, what objects does a window contain? Menus? Pixels? Areas? Color? What objects does an area contain? Pixels? Scan lines? Windows? This is just as simple as it seems. Just ask the question, “Does this object contain that object?” If you get immediate laughter, move on. Positive answers or thoughtful looks indicate a possible relationship.
3. Finally, determine what properties and behaviors each object class will have. These questions can be asked during the same user test as in step 2 because the answers users will give are closely related. Will you be able to ask windows for their names or pixels for their colors? How about asking windows to move or close? Can you ask pixels to change color or make a copy?

You may have noticed that this approach falls into the category of Good Object Design. Undoubtedly, anyone who does object-oriented design has gone through a similar process when developing an application. Resist the temptation to design the application’s internal structure using G.O.D. and be done with it, because the object model design is different from the application design. When designing the application, you typically analyze structure from the perspective of eventually implementing the design. Thus, you impose design constraints to make implementation easier. For example, you probably don’t keep representations of images, areas, and pixels, but choose one model for your internal engine — a reasonable solution for a programmer looking at the problem space. A typical image processing program usually has real classes representing images, and probably has an area class, but may not have a pixel class or scan line class. Pixels and scan lines may be implemented by a more basic representation than classes — simple indices or pointers into a PixMap, for example.

However, when you design object model support, you have a very different perspective. You’re designing classes based on user expectation and intention, not on programmer constraints. In object model design of an image processor, you *do* have TImage, TArea, TScanLine, and TPixel classes, regardless of your internal representation. This is because a user *sees* all these classes. The TImage and TArea may be the same as your internal engine’s TImage and TArea, and probably are. After all, there’s little reason to ignore a perfectly usable class that already exists. However, the TPixel and TScanLine classes exist only to provide object model support. I call classes that exist only to provide object model support *ephemeral* classes.

Undeniably, the most useful tool for finding objects is user testing. Another important source of information is the Apple Event Registry. The Apple Event Registry describes Apple event classes that are standardized in the Apple event world. The Registry lists each class along with its inheritance, properties, and behaviors. It’s also the last word on the values used to code object model support. For example,

The Apple Event Registry is on the *Developer CD Series* disc and is available in print from APDA (#R0130LL/A).•

constants for predefined Boolean operators and class types are listed in detail. As you follow the process for finding the objects in your application, you can use the elements found in the Registry as a basis for your investigation and for later implementation. For example, if your user tests reveal that a pixel class is appropriate for your application and a Pixel class is documented in the Registry, you should probably use the behaviors and properties documented there as a basis for your application's TPixel class. Doing so allows your application to work well with existing scripts that manipulate pixels and allows your users to have a consistent scripting experience across all pixel-using applications.

OSL CONCEPTS

In addition to the principles imposed by the object model itself, the OSL makes a few reasonable assumptions about what applications provide to support their objects. Since the object model requires that objects be able to retrieve contained objects, the OSL allows an object to count the number of objects of a given type contained within them. So, if an image contains scan lines, the image object needs to be able to count the number of scan line objects contained within it. Of course, in some circumstances, the number of objects that are contained can't be counted or is just plain big (try asking how many TSand objects are contained in a TBeach object). In this case, the OSL allows the object to indicate that the number can't be counted.

Additionally, the OSL allows applications to apply simple Boolean operators to two objects. The operators themselves are a part of the *Apple Event Registry.* They include the familiar operators like less than, equal to, and greater than as well as some more interesting relations like before, after, over, and under. The requirement for these operators is that they have Boolean results. This means that if *object1* and *object2* have *operator* applied to them, the expression *object1 operator object2* is either true or false. Of course, there's no requirement that every class implement every operator, only those that make sense. It makes little sense to ask if an object of type TColor is *greater than* another, but *brighter than* is another story.

During resolution of an Apple event, the OSL asks for tokens of objects between the application object and the final target to be returned (as described earlier in this article in "A Sample OSL Resolution"). To a programmer, they look like AEDescs being passed around, but the OSL treats them specially:

- The OSL guarantees that it will never ever look in the data portion of the token, the dataHandle field of the AEDesc. It may peek at the descriptorType field from time to time, but the data itself is golden. This becomes a critical point when applying the OSL engine to an object-oriented interface. The token data of Apple event objects should be "real" object references in whatever programming language is appropriate, and keeping the data completely private to the application makes this possible.

- The application must be able to recognize the token when it appears again. Thus, if the application returns a token for the image "Girl with Hat" to the OSL, the application must be able to recognize the significance of having that token passed back by the OSL.
- The OSL asks only that we guarantee the validity of a token during the resolution of the current object specifier.

Since the data contained in the AEDescs is private, the OSL must provide a system for the application to know when a token is being created and when it's being terminated. Creation of tokens is provided through the containment accessor protocol. Termination is provided by a callback routine which does the actual token disposal and which the application registers with the OSL. This callback is invoked from AEDisposeToken and comes in handy when applying the object model to C++ classes.

There are also a number of features that are beyond the scope of this article. One of these is the OSL concept of *marking* objects. This means that objects are labeled as belonging to a particular group. The contract the OSL makes with the application is that the OSL will ask whenever it needs a new kind of mark, and the application will recognize whether any object is marked with a particular mark. Further, given the mark itself, the application will be able to produce all the objects with that mark. If this sounds particularly confusing, just consider mark objects as typical list objects. Given a list and an object, it's quite natural to answer the question, "Is this object in this list?" Further, it's quite natural to answer the question, "What are all the objects contained in this list?"

The framework for adding Apple event support described later in the section "Inside UAppleObject" satisfies the basic OSL requests for counting objects, applying Boolean operators, and handling tokens. However, it doesn't handle marks. The intrepid reader could add support for this feature with a little thought.

CLASS DESIGN

To incorporate object model support into your applications, you need a class library that implements the object model classes you want to support — for example, the TWindow, TImage, TArea, and TPixel classes described earlier. These classes exist because they represent Apple event objects the application will support. Then you create a mapping of Apple event objects to the C++ classes that implement them (see Figure 3). For the sake of argument, say that TWindow, TArea, and TImage are also part of the class library used to implement the non–object-model portions of the program. The TPixel class is an ephemeral class. What these four classes have in common is a mixin class, MAppleObject, that provides the hooks for adding object model functionality (see the next section, "Inside UAppleObject," for more details).

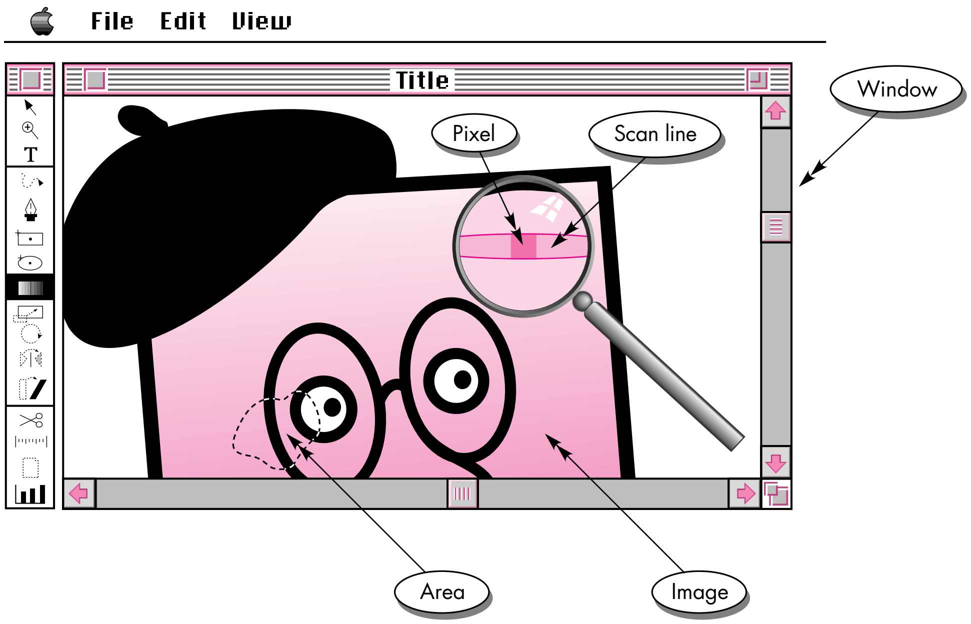


Figure 3
The Objects As Implemented

MAppleObject must include protocol that implements the object model and OSL concepts. Given an MAppleObject, there should be protocol for returning an object contained within MAppleObject. This accessor method is expected to return an object that satisfies the containment request. It also needs to inform the framework if the returned object is an ephemeral object — some might say that such an object is *lazy evaluated* into existence. As a practical matter, this informs the framework whether an object needs to be deleted when the OSL disposes of the object's token (as described in "A Sample OSL Resolution"). Obviously, it would be undesirable to have the framework delete the TImages because the application depends on them for its internal representation. It would be equally stomach-turning to have all the TPixels pile up in the heap, never to be deleted.

Since TPixel objects don't actually exist until they're lazy evaluated into existence, you're free to design their implementation in a wide variety of ways. Remember that one of the contracts the OSL makes with the application is that tokens need to be valid only during the resolution of the current object specifier. Well, consider that the implementation of images is just a handle of gray values. Normally, if someone suggested that a pixel be implemented as an index into a block of data, you'd throw temper tantrums. "What!" you'd yell, "What if the pixel is moved in the image! Now the index is stale." This is not an issue for tokens, because they're transient. Since pixels won't be added during the resolution of an object specifier, such a

The naming convention I use for classes differentiates between classes that are intended to be instantiated directly and those that are intended to be used as a mixin class. Classes that are directly instantiable begin with an uppercase *T* — TPixel, for example. Similarly, mixin classes begin with an uppercase *M* — MAppleObject, for example.•

TPixel objects don't actually exist until someone — usually the OSL — asks for them. Before that, pixels are hidden within other objects, probably TImage or TArea objects. However, when someone asks for a pixel object, suddenly a TPixel is *lazy evaluated* into existence.•

representation is fine. Of course, if you'd prefer a more robust implementation, that's fine, too, but remember that the OSL doesn't impose such robustness on you.

MAppleObject must also include a protocol to implement the comparison operators, counting protocol, and behavior dispatching. As a practical matter, these methods will likely be large switch statements that call other, more meaningful, methods depending on the details of the request. For example, the counting protocol might key on the kind of objects that should be counted and invoke methods specialized to count contained objects of a specific class.

Finally, each class provides protocol for telling clients which object model class the object represents. This is necessary for the framework to be able to communicate with the OSL. During the resolution conversation the OSL holds with the framework, the framework returns descriptors of each object the OSL asks for. These descriptors are required to publish to the OSL the type of the object returned from the request.

INSIDE UAPPLEOBJECT

UAppleObject is a framework whose main contribution is the class MAppleObject. MAppleObject provides the basis for integrating Apple event objects and Apple event object support into object-oriented applications. UAppleObject also includes a dispatcher, TAppleObjectDispatcher, and the 'aedt' resource. You drop the UAppleObject files into your application and immediately begin subclassing to provide Apple event functionality.

EXCEPTION HANDLING IN UAPPLEOBJECT

Developers familiar with the details of Apple event implementation are no doubt aware that the Apple Event Manager deals exclusively with error code return values, as does the rest of the Toolbox. When the Apple Event Manager invokes a developer-supplied callback routine, that routine commonly returns an integer error code. This style of error handling is found nowhere in UAppleObject. Instead, UAppleObject uses the UMAFailure unit to provide exception handling. UMAFailure is a unit available on the *Developer CD Series* disc that provides both a MacApp-style exception-handling mechanism for non-MacApp programs and excellent documentation for its use.

Wherever UAppleObject is invoked through a callback routine that expects an error code to be returned, all exceptions are caught and the exception's error code is returned to the Toolbox. Therefore, when an error occurs, call the appropriate FailXXX routine provided by UMAFailure — for example FailMemError, FailNIL, or FailOSErr. In the UAppleObject documentation, calling one of these routines is referred to as throwing an exception.

MAPPLEOBJECT

The major workhorse of UAppleObject is MAppleObject, an implementation of the basic Apple event object functionality. MAppleObject is an abstract mixin class that provides the protocol necessary for the UAppleObject framework to resolve Apple event objects and handle Apple events.

```
class MAppleObject
{
public:
            MAppleObject();
            MAppleObject(const MAppleObject& copy);
   virtual ~MAppleObject();

   MAppleObject& operator=(const MAppleObject& assignment);

   virtual DescType GetAppleClass() const = 0;

   virtual long CountContainedObjects(DescType ofType);
   virtual MAppleObject* GetContainedObject(DescType desiredType,
      DescType keyForm, const AEDesc& keyData, Boolean& needDisposal);
   virtual Boolean CompareAppleObjects(DescType compareOperator,
      const MAppleObject& toWhat);
   virtual void DoAppleEvent(const AppleEvent& message,
      AppleEvent& reply, long refCon);

   static void SetDefaultAppleObject(MAppleObject* defaultObject);
   static MAppleObject* GetDefaultAppleObject();

   static void GotRequiredParameters(const AppleEvent& theAppleEvent);

   static void InitAppleObject(TAppleObjectDispatcher* dispatcher = nil);
};
```

GetAppleClass

```
DescType GetAppleClass() const = 0;
```

GetAppleClass is an abstract method that returns the object model type of an object. Every MAppleObject subclass should override this method to return the object model type specific to the individual object.

CountContainedObjects

```
long CountContainedObjects(DescType ofType);
```

CountContainedObjects should return the number of objects of the indicated type that are contained within the receiver object. This is usually done by counting the number of objects your subclass knows how to access and adding it to the number of objects the parent class finds (in other words, call the inherited version and add it to the number you find yourself). If the number of objects is too large to be enumerated in a signed 16-bit integer, CountContainedObjects may throw the errAEIndexTooLarge exception.

GetContainedObject

```
MAppleObject* GetContainedObject(DescType desiredType, DescType keyForm,
   const AEDesc& keyData, Boolean& needDisposal);
```

GetContainedObject is a generic method for obtaining an object contained by the receiver. Subclasses always override this method to provide access to the subclass's contained objects. The desiredType, keyForm, and keyData arguments indicate the specific object to be returned as the function result. If the resulting object is one used in the framework of the application, GetContainedObject should return false in the needDisposal argument.

The alternative is for GetContainedObject to create the resulting object specifically for this request; in this case, it returns true in the needDisposal argument. If needDisposal is true, the UAppleObject framework deletes the result object when it's no longer needed.

CompareAppleObjects

```
Boolean CompareAppleObjects(DescType compareOperator,
   const MAppleObject& toWhat);
```

CompareAppleObjects performs the logical operation indicated by the arguments, returning the Boolean value of the operation. The semantics of the operation is *this compareOperator toWhat*. So, if the compareOperator parameter were kAEGreaterThan, the semantics of the method call would be *this is greater than toWhat*. Subclasses always override this method to provide the logical operations they support.

DoAppleEvent

```
void DoAppleEvent(const AppleEvent& message, AppleEvent& reply,
   long refCon);
```

When an object is identified as the target of an Apple event, it's sent the DoAppleEvent message. The message and reply Apple event records are passed in the corresponding arguments. If the direct parameter to the message is typeObjectSpecifier, the object specifier is guaranteed to resolve to the receiver; otherwise the receiver is the application object. Additional modifiers for the event can be extracted from the message, and the reply should be filled in by DoAppleEvent, if appropriate. The refCon parameter is the shortcut number registered with the UAppleObject framework (see the section "The 'aedt' Resource"). Subclasses always override DoAppleEvent to dispatch their supported Apple events to appropriate methods.

SetDefaultAppleObject and GetDefaultAppleObject

```
void MAppleObject::SetDefaultAppleObject(MAppleObject* defaultObject);
MAppleObject* MAppleObject::GetDefaultAppleObject();
```

GetDefaultAppleObject returns the MAppleObject currently registered as the null container. Similarly, SetDefaultAppleObject registers a particular object as the null container. Usually, the object serving as null container doesn't change during the lifetime of the application — it's always the application object. In this case, just call SetDefaultAppleObject from within your application object's constructor. But remember that any Apple event that arrives when no null container is registered falls on the floor and is returned to the Apple Event Manager with the errAEEventNotHandled error.

GotRequiredParameters

```
void MAppleObject::GotRequiredParameters(const AppleEvent&
   theAppleEvent);
```

GotRequiredParameters is here for convenience. To do Apple event processing "right," each Apple event handler should check that it has received everything the sender sent. Almost every good Apple event sample has this routine and calls it from within the handlers. Since all handling is done from within an MAppleObject method, it makes sense for this protocol to be a member function of MAppleObject. However, the member function really doesn't need access to the object itself, and could actually be called from anywhere, so it's a static member function.

InitAppleObject

```
void MAppleObject::InitAppleObject(TAppleObjectDispatcher* dispatcher =
   nil);
```

InitAppleObject must be called once after the application initializes the Toolbox and before it enters an event loop (specifically, before WaitNextEvent gets called). This method installs the given object dispatcher, or creates a TAppleObjectDispatcher if nil is passed.

TAPPLEOBJECTDISPATCHER

The second element of UAppleObject is TAppleObjectDispatcher. Together with MAppleObject, TAppleObjectDispatcher forms a complete model of Apple events, the objects themselves, and the Apple event engine that drives the object protocol. TAppleObjectDispatcher is responsible for intercepting Apple events and directing them to the objects that should handle them. A core feature of this engine is the ability to resolve object specifiers into "real" objects.

```
class TAppleObjectDispatcher
{
public:
   TAppleObjectDispatcher();
   virtual ~TAppleObjectDispatcher();

   virtual void Install();

   virtual MAppleObject* ExtractObject(const AEDesc& descriptor);
   virtual void StuffDescriptor(AEDesc& descriptor, MAppleObject* object);

   virtual void HandleAppleEvent(const AppleEvent& message,
      AppleEvent& reply, long refCon);

   virtual void AccessContainedObjects(DescType desiredClass,
      const AEDesc& container, DescType containerClass, DescType form,
      const AEDesc& selectionData, AEDesc& value, long refCon);
   virtual long CountObjects(const AEDesc& containerToken,
      DescType countObjectsOfType);
   virtual Boolean CompareObjects(DescType operation, const AEDesc& obj1,
      const AEDesc& obj2);
   virtual void DisposeToken(AEDesc& unneededToken);

   virtual MAppleObject* GetTarget(const AppleEvent& message);

   virtual void SetTokenObjectDisposal(MAppleObject* tokenObject,
      Boolean needsDisposal);
   virtual Boolean GetTokenObjectDisposal(const MAppleObject*
      tokenObject);

   virtual MAppleObject* ResolveSpecifier(AEDesc& objectSpecifier);

   virtual void InstallAppleEventHandler(AEEventClass theClass,
      AEEventID theID, long refCon);

   static TAppleObjectDispatcher* GetDispatcher();
};
```

Install

```
void Install();
```

Install is called when the dispatcher object is actually installed (at InitAppleEvent time). It's responsible for reading the 'aedt' resources for the application and declaring the appropriate handlers to the Apple Event Manager as well as registering with the OSL. Overrides should call the inherited version of this member function to maintain proper functionality. This method may be overridden to provide functionality beyond that supplied by TAppleObjectDispatcher — to provide for mark tokens, for example, which are left as an exercise for the reader. (Don'cha just hate it when articles do this to you?)

ExtractObject and StuffDescriptor

```
MAppleObject* ExtractObject(const AEDesc& descriptor);
void StuffDescriptor(AEDesc& descriptor, MAppleObject* object);
```

One of the key abstractions provided by TAppleObjectDispatcher is the packaging of MAppleObjects into tokens for communication with the Apple Event Manager and OSL. ExtractObject and StuffDescriptor are the pair of routines that carry the responsibility for translation. ExtractObject returns the MAppleObject contained within the token descriptor, while StuffDescriptor provides the inverse function. These functions are extensively used internally, but are probably of little interest to clients. Subclasses that override one method should probably override the other as well.

HandleAppleEvent

```
void HandleAppleEvent(const AppleEvent& message, AppleEvent& reply,
   long refCon);
```

HandleAppleEvent is called whenever the application receives an Apple event. All responsibility for distributing the Apple event to an object is held by this member function. HandleAppleEvent is rarely overridden.

AccessContainedObjects

```
void AccessContainedObjects(DescType desiredClass,
   const AEDesc& container, DescType containerClass, DescType form,
   const AEDesc& selectionData, AEDesc& value, long refCon);
```

At times during the resolution of an object specifier, MAppleObjects are asked to return objects contained within them. AccessContainedObjects is called when the parsing engine makes that query (in other words, it's the polymorphic counterpart of the OSL's object accessor callback routine). The method is responsible for getting the MAppleObject container, making the appropriate inquiry, and returning the result, properly packed. AccessContainedObjects is rarely overridden.

CountObjects

```
long CountObjects(const AEDesc& containerToken,
   DescType countObjectsOfType);
```

At times during the resolution of an object specifier, it may be helpful to find out how many of a particular object are contained within a token object. This method is called when the parsing engine makes that query (in other words, it's the polymorphic counterpart of the OSL's count objects callback routine). It's responsible for finding the MAppleObject corresponding to the token, making the inquiry of the object, and returning the answer.

CompareObjects

```
Boolean CompareObjects(DescType operation, const AEDesc& obj1,
   const AEDesc& obj2);
```

At times during the resolution of an object specifier, it may be helpful to compare two objects to determine if some logic relationship (for example, less than, equal to, before, or after) holds between them. CompareObjects is responsible for making the inquiry of the appropriate MAppleObject and returning the result (in other words, it's the polymorphic counterpart of the OSL's compare objects callback routine). The semantics of the operation is *obj1 operation obj2*. So, if the compareOperator parameter were kAEGreaterThan, the semantics of the method call would be *obj1 is greater than obj2*. This method is rarely overridden.

DisposeToken

```
void DisposeToken(AEDesc& unneededToken);
```

DisposeToken is called when the OSL determines that a token is no longer necessary. This commonly occurs during resolution of an object specifier. DisposeToken is responsible for acting appropriately (in other words, it's the polymorphic counterpart of the OSL's object disposal callback routine). For the implementation in TAppleObjectDispatcher, this means the routine checks to see if the object is marked as needing disposal, and deletes the object if necessary.

GetTarget

```
MAppleObject* GetTarget(const AppleEvent& message);
```

GetTarget is responsible for looking at the Apple event and determining which object should receive it. Notably, GetTarget is used by HandleAppleEvent. The TAppleObjectDispatcher implementation sends the Apple event to the default object unless the direct parameter is an object specifier. If the direct parameter is an object specifier, it's resolved to an MAppleObject, which is then sent the Apple event. This method is rarely overridden.

SetTokenObjectDisposal and GetTokenObjectDisposal

```
void SetTokenObjectDisposal(MAppleObject* tokenObject,
   Boolean needsDisposal);
Boolean GetTokenObjectDisposal(const MAppleObject* tokenObject);
```

Any MAppleObject can be marked as needing disposal or not needing it. SetTokenObjectDisposal and GetTokenObjectDisposal manage the internal representation of the table that keeps track of such information. You may want to override them both (never do it one at a time) to provide your own representation.

ResolveSpecifier

```
MAppleObject* ResolveSpecifier(AEDesc& objectSpecifier);
```

ResolveSpecifier returns the MAppleObject that corresponds to the object specifier passed as an argument. Under most circumstances, you don't need to call this routine since it's called automatically to convert the direct parameter of an Apple event into an MAppleObject. If, however, in the course of handling an Apple event, you find another parameter whose descriptorType is typeObjectSpecifier, you'll probably want to resolve it through this routine. Remember that objects returned from ResolveSpecifier may need to be deleted when the application is done with them. To accomplish this, you may either stuff the object into an AEDesc by calling StuffDescriptor and then call AEDisposeToken, or ask whether the object needs to be deleted by calling GetTokenObjectDisposal and delete it if true is returned.

InstallAppleEventHandler

```
void InstallAppleEventHandler(AEEventClass theClass, AEEventID theID,
   long refCon);
```

InstallAppleEventHandler is very rarely overridden. It's responsible for registering an Apple event with the Apple Event Manager, notifying the manager that the application handles the Apple event.

GetDispatcher

```
TAppleObjectDispatcher* GetDispatcher();
```

This static member function returns the dispatcher object that's currently installed. It's useful for calling TAppleObjectDispatcher member functions from a global scope.

THE 'AEDT' RESOURCE

The last piece of the UAppleObject puzzle is the 'aedt' resource. The definition of this resource type is in the Types.r file distributed with MPW. Developers familiar with MacApp's use of the 'aedt' resource already know how it works in UAppleObject because UAppleObject uses the same mechanism.

The 'aedt' resource is simply a list of entries describing the Apple events that an application handles. Each entry contains, in order, the event class, the event ID, and a numeric reference constant. The event class and ID describe the Apple event the application supports and the numeric constant is used internally by your application. The constant should be different for each supported Apple event. This allows your application to recognize the kind of Apple event at run time by looking at the refCon passed to DoAppleEvent.

When installed via the Install method, a TAppleObjectDispatcher object looks at all 'aedt' resources in the application's resource fork, registering all the Apple events in them. Thus, additional Apple event suites can be signified by adding resources

The TAppleObjectDispatcher implementation registers a static member function as the actual handler of the Apple event. This static member function calls the dispatcher's HandleAppleEvent method polymorphically. Thus, you'll most likely get the behavior you want out of an override of HandleAppleEvent.●

instead of adding to one resource. For example, the Rez code to define an 'aedt' resource for the four required Apple events is as follows:

```
resource 'aedt' (100) {{
   'aevt', 'oapp', 1;
   'aevt', 'odoc', 2;
   'aevt', 'pdoc', 3;
   'aevt', 'quit', 4;
}};
```

When the Open Document Apple event ('aevt', 'odoc') is sent to the application, the refCon value to DoAppleEvent is 2. Since you've assigned a unique numeric constant to each different Apple event, a refCon value of 2 can be passed to DoAppleEvent only when the Apple event is Open Document.

To add the mythical foobar Apple event ('foo ', 'bar ') to the application, mapped to number 5, you may either add a line to the resource described above or add another resource:

```
resource 'aedt' (101) {{
   'foo ', 'bar ', 5;
}};
```

EXTENDING CPLUSTESAMPLE

So far this sounds all well and good. The theory behind adding Apple event object support holds together well on paper. The framework, UAppleObject, has been written and works. The only thing left is to put my money where my mouth is and actually use UAppleObject to demonstrate the addition of Apple events to an Apple event–unaware application. The subject of this foray into the Twilight Zone is CPlusTESample in the Sample Code folder on the *Developer CD Series* disc. TESample serves as the basis for adding scripting support for object model classes.

CPlusTESample is attractive for a number of reasons. First, it's a simple application that could support some nontrivial Apple events. Second, it's written in an object-oriented style and contains a decent design from the standpoint of separating the user interface from the engine and internal representation. Finally, it's written in C++, a necessary evil for the use of UAppleObject.

To prove that CPlusTESample actually had the necessary flexibility to add Apple events, I began by adding font, font size, and style menus to the original sample. Adding these features required little modification to the original framework aside from the addition of methods to existing classes. Thus, I was satisfied that the underlying assumptions and framework could hold the paradigm shift of adding Apple event support.

UAppleObject is easier to implement in dynamic languages like Smalltalk or Macintosh Common Lisp. However, these packages don't yet lend themselves to creating commercial applications (no flames, please). The only language that has the requisite malleability and marketability is Uncle Barney's love child. Sorry, folks.•

In identifying the objects of the program, I chose windows and text blocks as the central object classes. If I were more gutsy, I would have attempted to actually define words and characters. However, the ancient programmer's credo crept in — it was more work than I was willing to do for this example. Further complicating this decision was the fact that CPlusTESample is built on TextEdit. Therefore, the obvious concepts of paragraphs and words translated exceptionally poorly into the internal representation, TEHandles. Characters would have been simpler than either paragraphs or words, but I copped out and left it as an exercise for the reader.

The relationships between classes are very straightforward. Windows are contained by the null object and text blocks are contained by windows. However, since I had a concept of window, it became interesting to define various attributes contained in windows: name, bounding box, and position. So, object model classes were defined for names, bounding boxes, and positions.

Behaviors were similarly straightforward. Text blocks, names, bounding boxes, and positions had protocol for getting their data and setting their data. Thus, an Apple event could change a name or text block or could ask for a position or bounding box.

In the end, six classes were defined to implement the object model classes: TESample, TEDocument, TWindowName, TWindowBounds, TWindowPosition, and TEditText. TESample is the application class and functions as the null object. TEDocument implements the window class and is used as the internal representation of the document and all its data. The remaining four classes are ephemeral classes that refer to a specific TEDocument instance and represent the indicated feature of that instance.

From that point, it was straightforward to write methods overriding MAppleObject to provide the containment, counting, comparison, and behavior dispatching. You can check out CPlusTESample with Apple event support added on the *Developer CD Series* disc.

IMPLEMENTING A CLASS

This section shows how UAppleObject helps you write cleaner code by looking at one of the CPlusTESample classes in detail — TEditText, the text class. User testing revealed the need for a class to represent the text found inside a CPlusTESample window, so I created a TEditText class whose objects are contained within some window class. Additionally, users wanted to retrieve and set the text represented by the text class. The *Apple Event Registry* defines a text class that roughly resembles the text class I wanted to provide in my CPlusTESample extension. Therefore, I decided to use the Registry's description as a basis for my TEditText class.

TEditText provides object model support for the user's concept of text, indicating that it should inherit from MAppleObject. TEditText objects don't contain any other objects, so there's no need to override the CountContainedObjects or GetContainedObject methods. However, TEditText objects do respond to Apple events. The Registry says that text objects should provide access to the text data itself through the Set Data and Get Data Apple events. Therefore, TEditText should include methods to implement each Apple event and should override DoAppleEvent to dispatch an Apple event to the appropriate method. After taking all this into account, here's what TEditText looks like:

```
class TEditText : public MAppleObject
{
public:
   TEditText(TEHandle itsTE);

   virtual void DoAppleEvent(const AppleEvent& message,
      AppleEvent& reply, long refCon);
   virtual DescType GetAppleClass() const;

   virtual void DoAppleGetData(const AppleEvent& message,
      AppleEvent& reply);
   virtual void DoAppleSetData(const AppleEvent& message,
      AppleEvent& reply);
private:
   TEHandle fTEHandle;
};
```

The constructor is relatively simple to implement. Since CPlusTESample uses TextEdit records internally, it's natural to implement TEditText in terms of TextEdit's TEHandle data structure. Therefore, TEditText keeps the TEHandle to which it refers in the fTEHandle instance variable.

```
TEditText::TEditText(TEHandle itsTE)
{
   fTEHandle = itsTE;
}
```

UAppleObject requires each MAppleObject instance to describe its object model class type through the GetAppleClass method. Since all TEditText objects represent the Registry class denoted by typeText, TEditText's GetAppleClass method is exceptionally straightforward, blindly returning the typeText constant.

```
DescType TEditText::GetAppleClass() const
{
   return typeText;
}
```

DoAppleEvent is also straightforward. It looks at the refCon parameter to determine which Apple event–handling method should be invoked. This method represents a large part of the remaining tedium for Apple event coding. Each class is responsible for translating the integer-based Apple event specifier, refCon in this example, into a polymorphic method dispatch such as the invocation of DoAppleSetData or DoAppleGetData. The nice part of this implementation is that subclasses of TEditText won't need to implement DoAppleEvent again if all the subclass needed was the Set Data or Get Data protocol. Instead such a subclass would simply override the DoAppleSetData or DoAppleGetData method and let the C++ method-dispatching mechanisms do the work.

```
void TEditText::DoAppleEvent(const AppleEvent& message,
   AppleEvent& reply, long refCon)
{
   switch (refCon)
   {
   case cSetData:
      this->DoAppleSetData(message, reply);
      break;
   case cGetData:
      this->DoAppleGetData(message, reply);
      break;
   default:
      MAppleObject::DoAppleEvent(message, reply, refCon);
      break;
   }
}
```

DoAppleGetData and DoAppleSetData are the Apple event–handling methods of the TEditText class. To developers familiar with the traditional Apple Event Manager interfaces, these methods are the UAppleObject equivalents of what the Apple Event Manager calls Apple event handlers. Each method follows a general pattern common to most remote procedure call protocols, of which Apple events are an advanced form.

First, the Apple event–handling method reads additional information from the message Apple event. The DoAppleGetData method doesn't happen to need any additional information because the entire meaning of the message is found in the identity of the Apple event itself. However, DoAppleSetData needs one additional piece of information — the text that should be stuffed into the object.

Next, the handler method calls GotRequiredParameters, passing the message Apple event as the sole argument. GotRequiredParameters ensures that the handler has retrieved all the information that the Apple event sender has sent. (For a discussion of why this is necessary, see *Inside Macintosh* Volume VI, Chapter 6.)

Third, the handler method will do whatever is necessary to perform the Apple event and create necessary reply data. The Get Data Apple event requires the TEditText object to fill the reply Apple event with the text it represents. Therefore, the DoAppleGetData method should retrieve the text contained in the TEHandle and pack it into an appropriate Apple event descriptor, putting that descriptor into the reply Apple event. In contrast to Get Data, the Set Data Apple event requires no reply, but does require that the text represented by the TEditText object be changed to reflect the text contained by the message Apple event. Thus, the DoAppleSetData method should contain code that sets the text contained in the object's TEHandle to the text retrieved from the message Apple event.

```
void TEditText::DoAppleGetData(const AppleEvent& message,
      AppleEvent& reply)
{
   // Note: This method uses no additional parameters.

   // Make sure we have all the required parameters.
   GotRequiredParameters(message);

   // Pack the text from the TEHandle into a descriptor.
   CharsHandle theText = TEGetText(fTEHandle);
   AEDesc      textDesc;
   HLock((Handle) theText);
   OSErr theErr = AECreateDesc(typeText, (Ptr) *theText,
      GetHandleSize((Handle) theText), &textDesc);

   // Unlock the handle and check the error code, throwing an
   // exception if necessary.
   HUnlock((Handle) theText);
   FailOSErr(theErr);

   // Package the reply.
   theErr = AEPutParamDesc(&reply, keyDirectObject, &textDesc);

   // Dispose of the descriptor we created and check the reply from
   // packaging the reply, throwing an exception if necessary.
   OSErr ignoreErr = AEDisposeDesc(&textDesc);
   FailOSErr(theErr);
}

void TEditText::DoAppleSetData(const AppleEvent& message,
   AppleEvent& /* reply */)
{
   // Get the text data descriptor from the message Apple event.
   AEDesc   textDesc;
   FailOSErr(AEGetParamDesc(&message, keyAETheData, typeText,
      &textDesc));

   // Make sure we have all the required parameters.
   GotRequiredParameters(message);

   // Use the data in the text descriptor to set the text of TEHandle.
   HLock(textDesc.dataHandle);
   TESetText(*textDesc.dataHandle, GetHandleSize(textDesc.dataHandle),
      fTEHandle);
   HUnlock(textDesc.dataHandle);

   // Dispose of the text descriptor we created above.
   OSErr ignoreErr = AEDisposeDesc(&textDesc);
}
```

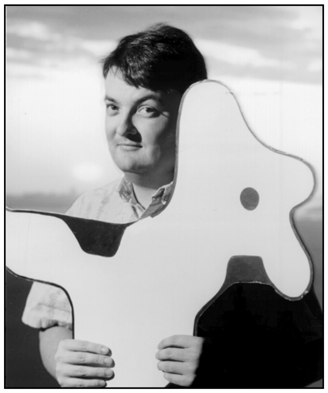
IT'S UP TO YOU

This article set out to reveal the deep significance of Apple events and the object model and to find a strategy for developing an object-oriented framework to take advantage of the Apple event object model design. Along the way, it danced around cognitive theory and discussed how cognitive theory applies to user perception of software. You've seen how object programming resembles such cognitive models to a more-than-trivial degree. And you've seen how those similarities can be leveraged to give workable, programmable models of user concepts within Turbo WhizzyWorks II NT Pro.

You've also seen the difficulties presented by the Apple Event Manager interface. Although Apple event objects and the object model are unarguably tied to user models and user-centric models, the Apple Event Manager is not. The UAppleObject framework presented here works with the object model and the Apple Event Manager to reduce generic user scripting to a tedious but straightforward task.

In the midst of all this detail, don't forget the payoff — providing a mechanism for users to interact with your applications using a level of control and precision previously undreamed of. The rest, as they say, is in your hands.

THANKS TO OUR TECHNICAL REVIEWERS
Richard Clark, C. K. Haun, Chris Knepper•



PETE ("LUKE") ALEXANDER

PRINT HINTS

TOP 10 PRINTING MISDEMEANORS

In my last column (in *develop* Issue 10), I talked about the "Top 10 Printing Crimes" that would cause you and your application serious headaches during print time. Here I'll list the "Top 10 Printing Misdemeanors." A printing misdemeanor will cause minor to major printing problems on different devices. Usually, you'll be able to get output onto a page, but it won't necessarily be what you want or where you want it.

Here's the list:

10. Using CopyMask and CopyDeepMask with the LaserWriter.
9. Using the obsolete spool-a-page, print-a-page method.
8. Not being very careful when using SetOrigin with the LaserWriter.
7. Creating pictures while the Printing Manager is open.
6. Not having all your data ready for the Printing Manager when you open it.
5. Making assumptions about the imageable area.
4. Using variables from Laser Prep (that is, **md**).
3. Checking wDev for the wrong reasons.
2. Accessing print record fields that are used internally.
1. Adding printing to your application four weeks before going final.

Most of these misdemeanors are easily avoided if you plan ahead. Let's take a look at the problems and the solution to each one.

SOLUTIONS TO THE MISDEMEANORS

10. Using CopyMask and CopyDeepMask with the LaserWriter.

It's not possible to directly print to a LaserWriter an image that was created with CopyMask or CopyDeepMask, because these calls aren't saved in pictures and they don't go through the stdBits QuickDraw bottleneck. The image's data must be recorded in the picture or go through the stdBits bottleneck in order for the LaserWriter driver to be able to image the data on the printer.

Solution: You can create your image in an off-screen world using CopyMask and CopyDeepMask to your heart's content. When you're ready to print your image, CopyBits it directly to the LaserWriter's grafPort using srcCopy.

9. Using the obsolete spool-a-page, print-a-page method.

There are still a few applications using the spool-a-page, print-a-page method of printing a document. This approach is no longer required unless you're printing from a Macintosh that doesn't have a hard drive. Otherwise, it's a bad idea; it has major drawbacks in the areas of speed and user happiness.

The idea of this method was to print each page of a document as a separate job. This was required in the early Macintosh days because disk space was at a premium. It prevented a document from filling up the entire disk and never printing a page. But in this age of hard disks, it's no longer needed.

Opening and closing the Printing Manager for each page could result in a serious speed penalty. And it

PETE ("LUKE") ALEXANDER Luke's latest adventure was landing his sailplane close to the edge of the earth (there's an actual sign, near Gerlach, north of Reno, that reads "The Edge of the Earth, 8 miles — Planet X"). Not only is this in the middle of nowhere, but rumor has it that Gerlach is the home of the best ravioli in Nevada. Luke and his friends didn't locate the ravioli, but as a consolation prize they stumbled onto Planet X instead (and Planets Y and Z, all art galleries, run by a slow-motion hippie who will reluctantly take MasterCard, if you have all year). The edge of the earth did deliver some great camping under the stars, and real cool satellite watching.•

could make your users very unhappy when printing to a shared printer; it's possible to have another user grab the printer before you do, thereby intermixing your pages with theirs.

Solution: Don't use the spool-a-page, print-a-page technique. Instead, use the method described in the Technical Note "A Printing Loop That Cares . . .".

8. *Not being very careful when using SetOrigin with the LaserWriter.*

If you're using SetOrigin to change the coordinate system when sending direct PostScript™ code to the LaserWriter, you'll run into trouble when printing in the foreground versus the background.

The PostScript LaserWriter drivers 4.0 through 5.2 handle SetOrigin differently when background printing is enabled.

- When background printing is disabled and the application calls SetOrigin, QuickDraw responds by adjusting the portRect of the printer driver's grafPort. Since SetOrigin doesn't cause any grafProcs to run (because no drawing occurs), the printer driver doesn't see the effect of this call until the next QuickDraw call is made (for example, DrawString or LineTo). At this point, the driver notices the change in the portRect and updates its internal origin. From then on, all QuickDraw and PostScript graphics are localized to the new origin.
- When background printing is enabled, QuickDraw is playing back a picture that was spooled earlier. When SetOrigin is encountered while DrawPicture is playing the picture, the grafPort's portRect isn't updated. Instead, QuickDraw keeps the current origin cached and offsets each graphic on the fly. Since the portRect wasn't modified, the printer driver doesn't see the SetOrigin call. Although all QuickDraw objects are still localized correctly (by QuickDraw), PostScript graphics don't move to the new origin.

In LaserWriter drivers 6.0 and later, the call to SetOrigin is a problem only on the first page that's spooled. After the first page, the driver looks at the grafPort's coordinates and then records the SetOrigin information correctly by inserting a picture comment into the spool file. This enables PrintMonitor to realize when the origin changes. Unfortunately, the driver never records the changes produced by a SetOrigin call when it's in the stdBits QuickDraw bottleneck.

Solution: In general, using SetOrigin doesn't buy you much, and it can get you in a lot of trouble. There are still a few printer drivers that don't handle the call correctly. Avoid using SetOrigin if possible.

If you use SetOrigin when sending direct PostScript code, use the techniques described in the Technical Note "Position-Independent PostScript" to ensure that all the PostScript code your application creates is position independent. To get the LaserWriter driver to realize as soon as possible that you've changed the coordinate system, you can send the following code:

```
PicComment (PostScriptBegin, 0, nil);
PicComment (PostScriptEnd, 0, nil);
```

This is a little weird, but it works because the two PicComment calls go through the stdBits QuickDraw bottleneck, which is where the driver checks and updates the coordinate system as required.

7. *Creating pictures while the Printing Manager is open.*

Some applications use a picture to collect all their QuickDraw objects before sending them on to the printer. This approach is OK unless the Printing Manager has already been opened by a call to PrOpen. The most noticeable problems are memory use and floating picture comments.

The memory problem can be very evident if you're printing to a printer driver that requires a lot of memory. Between your memory use and the printer driver's, there might not be enough memory available to meet everyone's appetite. Remember, there isn't a magical amount of memory that will guarantee that your application will print successfully.

The other significant problem you might encounter is floating picture comments. When this occurs, the picture comments sent by your application will be recorded out of order, which will usually cause your image to print its objects out of order.

Solution: Read the Technical Note "Pictures and the Printing Manager" before you start to use pictures at print time. Better yet, don't create a picture when the Printing Manager is open.

6. Not having all your data ready for the Printing Manager when you open it.

There aren't too many things you can do to speed up printing, but having data ready for the Printing Manager when you open it is one of them. If you open the Printing Manager and then go off to collect data you want to print, your printing time could increase dramatically. You also run the risk of timing out the print job because you don't send data to a networked printer fast enough or your print job takes too long to complete.

Solution: When you open the Printing Manager, have all your data collected and ready to send to the printer. Make sure the data is formatted for the current printer (see the next misdemeanor for additional details).

If your application needs to perform a lot of data collection or preparation (as would a database application), consider spooling all your information to disk as pictures. This is especially useful when you don't know how long it will take to gather the data for a particular page. To use this approach, you would open up a file and write out each page as a picture (as the Printing Manager does), spool everything to disk, and then send the pictures to the printer driver. Printing will be really fast! But be sure not to commit misdemeanor 7 above, and note that this should not be the only way your application prints; since you may not have enough disk space, you should make it an option in a Preferences or Print dialog.

Having your data ready to go when you open the Printing Manager ensures that you'll print as fast as possible and avoid timeout problems. And it will make your application a friendly networked printer user, compared to grabbing the printer on the network and hogging it while your application collects data.

5. Making assumptions about the imageable area.

Some applications make assumptions about the imageable area (the page rectangle) at print time. This can cause some serious speed and clipping problems. If any part of your image (which may contain text, QuickDraw objects, bitmaps, or pixMaps) falls outside the page rectangle, the printer driver will need to clip it. This will slow down the printing process and you won't get the output you want. The imageable area for each printer is slightly different; this is actually a good thing, since it allows the printer driver to take full advantage of the printer's capabilities.

About half of the printing game is reformatting your image to work for the currently selected printer. This problem is most noticeable when you print to a film recorder an image that was set up for a LaserWriter. If you don't reformat the image, you won't get the results you want; because of the higher resolution of the film recorder (1500 versus 300 dpi), you'll get a micro-image and you'll waste film. Also, most film recorders print only in landscape orientation.

Solution: Since each printer has a slightly different imageable area, you should format your image to this area. Before sending your data to the printer, you should format it to rPage, the page rectangle for the current printer. rPage lives in the TPrInfo record within the print record. However, be careful; as mentioned in the previous misdemeanor, you should have all your data ready to send (including all formatting) before opening the Printing Manager. Open the Printing Manager, get the dimension for rPage, close the Printing Manager, format your data, open the Printing Manager again, and print.

One approach for saving your data within your application to help you format it at print time is to specify the location of each object on the page as a percentage of distance (as opposed to pixels). For example, you could specify an object to be 10% from the top and left margins. You would then always be able to place the object in the correct position for all printers no matter what the resolution.

4. Using variables from Laser Prep (that is, md).

Using operators from the LaserWriter driver's dictionary **md** is a classic way of causing your application compatibility problems when a new LaserWriter driver is released. Some developers do this to achieve additional PostScript functionality at print time. The problem is that when Apple releases a new LaserWriter driver it usually changes a few of the operators in **md**. This will then break code that depends on **md**. It's an even bigger problem if you save this information in pictures. When a new LaserWriter driver is released, none of these pictures created by your users will be able to be printed.

Solution: Don't use any of the operators defined within **md** in your printing code. This has been around for a long time as a compatibility issue; take a look at the Technical Note "Using Laser Prep Routines" for the historical data.

If you decide to jump off the cliff and use operators in **md**, you owe it to your users to check the existence of an operator before you use it. This piece of PostScript code will do the trick:

```
userdict /md known
{
   md /bu known {myBU} if
} if
```

In this example, we're checking for the existence of **bu** before we replace it with our newly defined operator, **myBU**. If the **bu** operator didn't exist, we'd do the right thing (that is, we'd still be able to print).

3. Checking wDev for the wrong reasons.

The printer type (such as LaserWriter or StyleWriter) is stored as an unsigned char in the high byte of the print record's wDev field (in the TPrStl record). Each printer driver has a unique wDev, and there are now over 142 wDevs in the world. That's quite a few printers available for your application to print to.

If you're checking wDev to see which type of printer you're talking to, you could end up very disappointed. Relying on wDev to make decisions at print time makes your application completely device dependent. What do you do when you get a wDev you don't know about? You have to make assumptions about the printer, and if you make a bad decision, you won't get the output you expect. This isn't fair to your users; they should be able to print to any printer that's connected to the Macintosh.

When we were developing the StyleWriter printer, we had some serious compatibility problems with a few of the major applications. They assumed that any device with a resolution greater than 300 dpi must be a PostScript printer. They sent only PostScript code to the StyleWriter, which didn't work out too well, since of course the StyleWriter doesn't understand PostScript.

Solution: Don't check wDev, with a couple of exceptions. One exception is that you should check wDev and the printer driver version if you need to work around a bug in the printer driver. This is the only method available to determine whether you're dealing with a particular printer driver. Checking the driver version by calling PrDrvrVers is important, because when the bug is fixed, you can remove your fix and let the driver do the work. Another exception is that you can check wDev after you've created a valid print handle (by calling PrintDefault) to see if the user has changed the printer type (for example, a LaserWriter to a StyleWriter) via the Chooser. In any case, be sure that when you do check wDev, you check it as an unsigned char value.

2. Accessing print record fields that are used internally.

You may notice that this is similar to the number 2 printing crime in the Print Hints column in Issue 10. There I emphasized the crime of accessing private

("PT") fields that you may come across when prowling around in the print record. Also likely to cause inconsistent results is the misdemeanor of accessing other fields in the print record that are used internally (or unused). To make this even clearer, I'll tell you just what print record fields you *can* read and write.

The print record is chock full of information. It's an application's playground during printing. It's also used by printer drivers to hold information about the current print job. Since each printer has slightly different needs, each one uses these fields differently. The public API documented in *Inside Macintosh* is the same, but the rest of the print record is free domain for the printer driver to use as it sees fit.

Setting a field that the printer driver doesn't expect you to touch can cause big problems for your application. This is one of the reasons why printer drivers have compatibility problems when they're being developed, and why they take so long to create.

Solution: Don't set any fields in the print record besides iLstPage, iFstPage, pIdleProc, pFileName, and iFileVol. If you do, you're running a serious compatibility risk with new printer drivers and printers you don't have access to during your test cycle. See the Technical Note "A Printing Loop That Cares . . ." for details about setting and using iLstPage and iFstPage, and the Technical Note "Me and My pIdle Proc (or how to let users know what's going on during print time . . .)" for details about setting pIdleProc.

Don't read any fields in the print record besides the ones you can set and the fields rPage, rPaper, iCopies, iVRes, iHRes, bjDocLoop, and bFileVers. (You can also read the TPrStatus record returned by prPicFile.)

1. Adding printing to your application four weeks before going final.

This too is similar to a printing crime in Print Hints in Issue 10 — but there has been a change, to four weeks instead of two. I can't emphasize this enough. Since my last column, a couple of developers have come to us with major printing problems and a shipping deadline only a few weeks away. They had just started to add printing to their applications.

Solution: Designing printing at the beginning — not the end! — of your application's development cycle is the solution to most of your printing headaches. Printing performance can make or break an application. You should convince the right people in your organization that printing is just as important as any other feature. There are a few pitfalls in the current printing architecture, but most of these problems can be avoided without a lot of work — if you design printing into your application from the start.

So please, stay out of trouble and avoid the printing crimes and misdemeanors. You'll be a happy printing developer and your users will also be delighted.

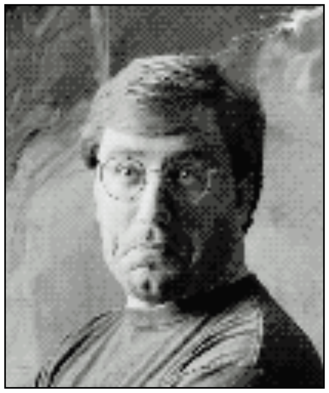
REFERENCES

- *Inside Macintosh* Volume II (Addison-Wesley, 1985), Chapter 5, "The Printing Manager," pages 150–151.
- "Print Hints: Top 10 Printing Crimes" by Pete ("Luke") Alexander, *develop* Issue 10.
- "Print Hints From Luke & Zz: CopyMask, CopyDeepMask, and LaserWriter Driver 7.0" by Pete ("Luke") Alexander, *develop* Issue 8.
- Macintosh Technical Notes "The Effect of Spool-a-page/Print-a-page on Shared Printers" (formerly #125), "Using Laser Prep Routines" (formerly #152), "A Printing Loop That Cares . . ." (formerly #161), "Position-Independent PostScript" (formerly #183), "Me and My pIdle Proc (or how to let users know what's going on during print time . . .)" (formerly #294), and "Pictures and the Printing Manager" (formerly #297).

Thanks to Hugo Ayala, Dave Hersey, and Scott ("Zz") Zimmerman for reviewing this column, and to Ana Wilczynski for the column idea.•

ANOTHER TAKE ON GLOBALS IN STANDALONE CODE

While MPW is great for developing applications, it provides little support for creating standalone code resources such as XCMDs, drivers, and custom window, control, and menu definition procedures, especially if you have nonstandard needs. Two roadblocks developers immediately notice are the inability to create more than 32K of object code and the lack of access to global variables. This article addresses the latter issue.



KEITH ROLLIN

The Macintosh Technical Note "Stand-Alone Code, *ad nauseam*" (formerly #256) does an admirable job of explaining what standalone code is and discussing the issues involved in accessing global variables from within it. I'll describe the solution proposed in that Tech Note later in this article, but you may also want to look over the Note before reading further.

It's important to realize that the Tech Note discusses just one possible solution to the problem of using global variables in standalone code. This article presents another solution, in the form of the StART package included on the *Developer CD Series* disc. Along the way, I'll talk a bit about what the issues are, describe how users of Symantec's THINK environments address the problem, recap the solution presented in the Tech Note, and show how to use MPW to implement a THINK-style solution. I'll also take a look at the advantages and disadvantages of each approach, allowing you to choose the right solution for your needs.

Note that the StART package is a solution for MPW users and that it assumes a lot about how MPW currently works. It's possible that you may not be able to use the StART package to develop standalone code that uses globals with future versions of MPW, although code already created with StART will, of course, continue to work.

WHAT IS STANDALONE CODE?

Standalone code is merely executable code that receives little to no runtime support from the Macintosh Operating System. The advantage of standalone code resources

KEITH ROLLIN is one of Taligent's charter members, sporting the obligatory snide business title of Phantom Programmer (he got this title after buying that lakefront property in the fifth basement of the Grand Opera House in Paris). When not fending off people asking him what he does at Taligent, Keith skis, rides his bike, reads voraciously, watches 1940s movies at the local oldies theater, and comes up with reasons not to shave. Look for his latest book, *Macintosh Programming Secrets,* 2nd edition, co-authored with Scott Knaster, at your local bookstore (he needs the money).•

is that they can be quickly loaded into memory, executed, and dismissed without the overhead of setting up a full-fledged runtime environment for them. In addition, standalone code can execute without affecting the currently running application or relying on it for any services. This makes such resources ideal for easily extending the system's or your application's functionality. By creating the right kinds of standalone code resources, you can change how controls or windows appear, or you can dynamically extend the capabilities of your application.

Table 1 shows a list of the most common system- and application-defined standalone code resources.

Table 1
Kinds of Standalone Code Resources

Resource Type	Resource Function
* `ADBS`	ADB device driver
* `adev`	AppleTalk link access protocol
`boot`	Boot blocks
`CACH`	System RAM cache code
* `CDEF`	Custom control definition
* `cdev`	Control panel device
* `dcmd`	Debugger extension
`dcmp`	Resource decompressor
* `DRVR`	Device driver
* `FKEY`	Function key
`FMTR`	3.5-inch disk formatting
* `INIT`	System extension
`itl2`	Localized sorting routines
`itl4`	Localized time/date routines
* `LDEF`	Custom list display definition
* `MBDF`	Custom menu bar definition
* `MDEF`	Custom menu definition
* `mntr`	Monitors control panel extension
`PACK`	System package
* `PDEF`	Printer driver
`PTCH`	System patches
`ptch`	System patches
* `rdev`	Chooser device
`ROvr`	ROM resource override
* `RSSC`	Resource editor for ResEdit
`SERD`	Serial driver
* `snth`	Sound Manager synthesizer
* `WDEF`	Custom window definition
* `XCMD`	HyperCard external command
* `XFCN`	HyperCard external function

Note: Items marked with an asterisk are ones that you might create for your own application, extension, driver, or whatever. The rest are reserved for the system.

Standalone code differs from the executable code that makes up an application, which has a rich environment set up for it by the Segment Loader. Let's take a look at an application's runtime environment so that we can better understand the limitations we must overcome to implement standalone code.

An application runs in a section of memory referred to as its partition. Figure 1 shows the layout of an application partition. A partition consists of three major sections. At the top of the partition is the application's *A5 world*, consisting of the application's global variables, the jump table used for intersegment function calls, and 32 bytes of application parameters (see "Application Parameters"). This area of memory is called the A5 world because the microprocessor's A5 register points into this data and is used for all access to it. Immediately below the A5 world is the *stack*, the area of memory used to contain local variables and return addresses. The stack grows downward toward the *heap*, which occupies the rest of the partition. The heap is used for all dynamic memory allocation, such as blocks created by NewHandle and NewPtr. Everything we see in Figure 1 — the heap (with a valid zone header and trailer), the stack, and the filled-out global variables and initialized jump table — is created by the Segment Loader when an application is launched.

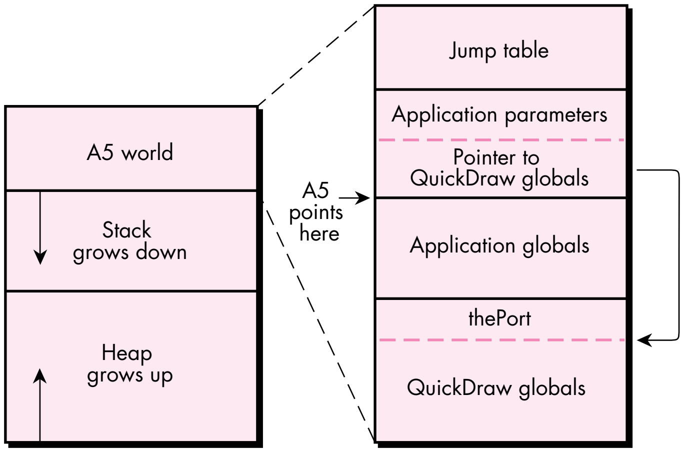


Figure 1
An Application Partition

This is the application's domain, and none shall trespass against it. And therein lies the conflict between applications and standalone code: Executing code needs to use the A5 register to access its global variables, but an application's use of A5 prevents any standalone code from using it with impunity. Additionally, the A5 world is created by the Segment Loader when an application is launched. Since standalone code is not "launched" (instead, it's usually just loaded into memory and JSRed to), it doesn't get an A5 world, even if A5 were available. We must solve these two problems

APPLICATION PARAMETERS

Not much is known about the mysterious 32 bytes directly above A5 known as application parameters. Figures 9 and 10 on pages 19 and 21 of *Inside Macintosh* Volume II indicate their existence, but the description simply says that "they're reserved for use by the system." We know that the first four bytes contain a pointer into the QuickDraw globals, but that's about it. Some MPW glue routines use some of the other bytes, but that use is undocumented. In any case, the application parameters seem pretty important. As you'll see later, we make sure our standalone code resources support them.

— the contention for A5 and the need to set up some sort of global variable space — in order to use globals in standalone code.

THE THINK SOLUTION

For years, users of THINK C and THINK Pascal have been able to use global variables in their CDEFs, LDEFs, drivers, and other types of standalone code. THINK has solved the problem of A5 contention by compiling standalone code to use the A4 register for accessing globals, leaving A5 untouched. Their solution to the need to set up global variable space is simply to attach the globals to the end of the standalone code, again leaving the application's A5 world untouched.

Figure 2 shows how standalone code created by a THINK compiler looks, both on disk and in memory. If the code was created with the C compiler, which allows preinitialized global variables, the global variable section contains the initial values. If the code was generated by the Pascal compiler, which sets all global variables to zero, the entire global section simply consists of a bunch of zeros (kind of like some guys I used to know in high school).

This is in contrast to the way globals are stored on disk for applications. MPW, for instance, uses a compressed data format to represent an application's globals on disk. When the application is launched, a small bit of initialization code is executed to read the globals from disk, expand them, and write them into the application global variable space in its A5 world.

Standalone code created by a THINK compiler accesses global variables by using A4-relative instructions. Because the use of the A4 register is ungoverned, such standalone code must manually set up A4 so that it can be used to reference its global variables. This setup is done by some macros provided by the THINK headers: RememberA0 and SetupA4. (It's called RememberA0, and not RememberA4, because the macro has to store the value in the A0 register temporarily.) When the standalone

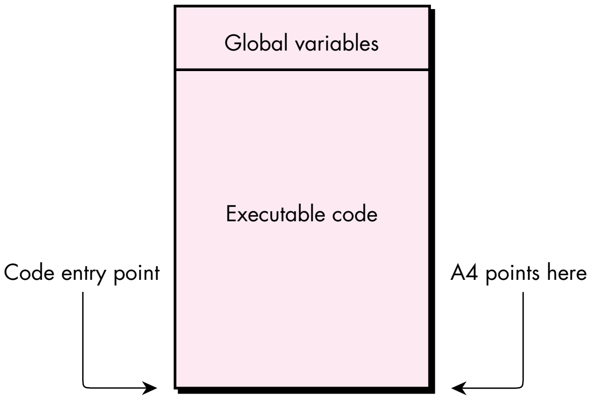


Figure 2
Format of a Standalone Code Resource Created by a THINK Compiler

code is finished and is about to return to its caller, it must call RestoreA4 to restore the value that was in A4 before the standalone code was called.

The solution provided by THINK offers many advantages:

- It's simple to use. Making sure you surround the entry point of your standalone code with the appropriate macros is easy, and the macros don't require any tricky parameters. Just type them in and you're done.
- The THINK development systems automatically insert a little bit of magic code at the beginning of standalone code resources that make the setting up of A4 as transparent as possible.
- THINK's use of A4 means that A5 is totally undisturbed, and hence A5 continues to point to a valid A5 world with, presumably, an initialized set of QuickDraw globals. This means that standalone code can make Toolbox calls without a second thought (or even much of a first thought, for that matter).
- Because the globals are attached to the standalone code, when the memory allocated to the standalone code resource is disposed of (for example, when the process that loaded it calls ReleaseResource on the segment), the globals are removed as well.

There are at least three disadvantages to THINK's approach, however:

- Since A4 is now pulling duty as the global variable reference base, fewer registers are available for calculating expressions, caching pointers, and so on. This means that the code generated is less efficient than if A5 were used for referencing globals.

For the sake of brevity, I occasionally refer to both the THINK C and THINK Pascal compilers simply as "THINK."•

- The globals are stored on disk in an uncompressed format, a fact you should be aware of before cavalierly declaring those empty 20K arrays.
- The resources holding the standalone code must not be marked as purgeable, or the global variables will be set back to their original values when the resource is reloaded.

A fourth disadvantage could be that the combined size of the executable code and the global variables must be less than 32K. However, this is somewhat ameliorated by THINK's support of multisegmented standalone code.

THE TECH NOTE SOLUTION

Users of THINK development systems have their solution for accessing global variables in standalone code. MPW users, however, don't have an immediately obvious solution. First, MPW's compilers don't have the option of specifying that A4 should be used to access global variables. Second, the MPW linker is written to create a compressed block of data representing the global variables and to place that block of data off in its own segment. Because A4 can't be used to access globals, and because the globals aren't attached to the end of the standalone code resource, MPW users don't have the slick solution that THINK users do.

A possible alternative was presented to MPW users a couple of years ago with the publication of the Technical Note "Stand-Alone Code, *ad nauseam*." Let's take a quick look at that approach, and then compare it with THINK's solution.

Let's start by examining the format of a simple application, shown in Figure 3. This is the format that MPW is designed to create, with any deviance from the standard formula being cumbersome to handle.

This application has three segments. CODE 0 contains the information used by the Segment Loader to create the jump table, the upper part of an application's A5 world. CODE 1 contains executable code, and usually contains the application's entry point. CODE 2 contains the compressed data used to initialize the global variable section of the application's A5 world, along with a little bit of executable code that does the actual decompressing. This decompression code is automatically called by some runtime setup routines linked in with the application. The purpose of the call to UnloadSeg(@_DataInit) in MPW programs is to unload the decompression code along with the compressed data that's no longer needed.

The solution proposed in the Tech Note is to use a linker option that combines segments 1 and 2. At the same time, the Note provides a couple of utility routines that create a buffer to hold the global variables and that decompress the variables into the buffer. Figure 4 shows what standalone code looks like when it's running in memory.

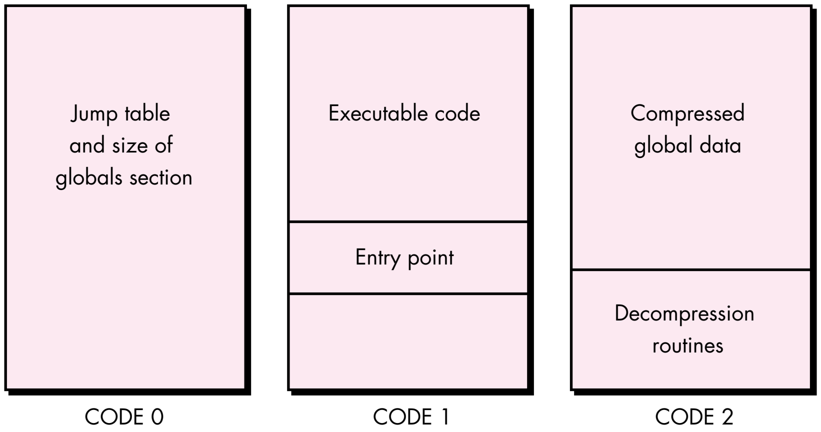


Figure 3
Format of a Simple Application Created by MPW

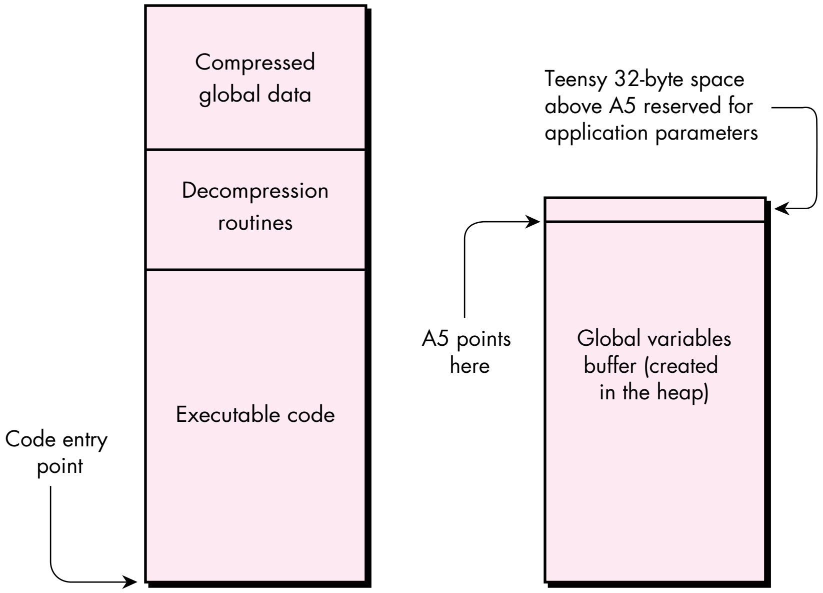


Figure 4
Format of Standalone Code Using the Tech Note Method

When the standalone code is called, it's responsible for creating and initializing its own A5 world. It does this by calling OpenA5World, which is directly analogous to THINK's SetupA4 macro. OpenA5World creates the buffer shown on the right in Figure 4, sets A5 to point to it, and calls the decompression routines to fill in the buffer. When the standalone code is ready to exit, it must call CloseA5World to deallocate the buffer and restore the original value of A5.

Note that this approach has an immediate disadvantage compared to the THINK approach. Because the global variables buffer is deallocated when the code exits back to the caller, all values that were calculated and stored in global variables are lost. This makes the OpenA5World/CloseA5World solution good if you simply want to use global variables in lieu of passing parameters, but lousy if you're trying to maintain any persistent data.

Fortunately, the Tech Note also presents a slight variation on the above solution that doesn't require that the global variables buffer be deallocated when the standalone code exits. However, the solution requires a little help from the host application. When the standalone code exits, it has two problems to consider. The first is that it must find some way to maintain a reference (usually a handle) to the buffer holding the global variables. After all, where can the standalone code store this reference itself? It can't store it in a global variable, because this reference will later be used to recover our global variables buffer. It can't store the reference in a local variable, because local variables are destroyed once the function that declares them exits.

The second problem that must be solved when creating a solution that doesn't require flushing the global variables is that of knowing when it actually is time to dispose of them. Globals accessed by THINK code resources are attached to the segments themselves, which means that they're disposed of at the same time as the code resource itself. What happens if the caller of a standalone code resource created using the OpenA5World technique decides that it no longer needs that resource? If it simply calls ReleaseResource on the resource, the global variables used by the standalone code will be stranded in the heap. This is known as a memory leak, and it is very bad. The block of memory holding the global variables is no longer referenced by any code, and there's no way to recover a reference to them. That block of memory will never be disposed of and will waste memory in the heap.

The approach that the Tech Note takes to solving both of these problems is to require the help of the caller (usually the host application). First, the caller must agree to maintain the reference to the standalone code's global variables buffer. After the buffer is created, the reference to it is passed back to the caller. The next time the standalone code is called, and all subsequent times, the caller passes that reference back to the standalone code, which then uses that reference to recover its global variables and reset A5 the way it likes it. Additionally, the caller must agree to notify the standalone code when it's about to go away. When the standalone code receives that notification, it takes the opportunity to dispose of the global variables buffer.

Our brief recap of the Tech Note outlines a workable approach that provides a few advantages over the solution provided by THINK:

- The on-disk representation of the standalone code is usually smaller, because the combination of the compressed data and decompression routines of MPW is often smaller than the raw data generated by THINK.
- Because the executable code and global variables are allocated in their own buffers, each of which can be 32K in length, you can create larger code resources and define more global variables. (This does not take into account the partial advantages provided by THINK's multisegmented standalone code.)
- Because MPW doesn't use it to access the globals, the A4 register can be used to generate more efficient object code.
- Since the globals are stored separately from the standalone code, the resource holding the standalone code can be marked as purgeable.
- The two blocks of memory holding standalone code and global variables can be locked or unlocked separately from each other, providing greater memory management flexibility.

There are, however, some disadvantages to the OpenA5World approach. The major disadvantage concerns the persistence of the global variables buffer. Either this buffer must be deallocated every time the code resource is exited, or the help of the caller must be elicited to maintain the reference to the buffer and to tell the standalone code when the buffer must be deallocated. If you're not in a position to define the responsibilities of the caller (for instance, if you're writing a WDEF), this disadvantage could be quite serious.

The second disadvantage concerns the reuse of the A5 register. Once the standalone code changes A5 from pointing to the caller's A5 world to pointing to the standalone code's globals, A5 no longer points to a valid set of QuickDraw globals. This can easily be solved by calling InitGraf early in the standalone code, but some problems may still exist. For instance, what if the standalone code needed to draw something in the current port (as an LDEF would need to do)? The GrafPtr of the port to be used is back in the caller's A5 world. Once we switch over to the standalone code's A5 world, we no longer know what port to draw into. This problem is briefly alluded to in the Tech Note, but it's not directly addressed.

THE START SOLUTION

It's possible to combine the advantages of the two approaches we've seen so far, while at the same time eliminating some of the disadvantages. The idea behind the hybrid approach I'll now present is to con MPW into creating a standalone code resource that has the same layout as one created by THINK. Specifically, instead of being stored in a separate buffer, the globals will be tacked onto the end of the code resource. This eliminates much of the reliance the standalone code has on the caller, and, as you'll see later, still allows us to create 32K worth of object code and 32K of global data.

As we saw when discussing the Tech Note approach, we need to get MPW to take the stuff it normally puts in an application and convert it to a standalone code resource. The OpenA5World solution used a linker option to accomplish this. My solution uses a custom MPW tool instead.

Let's begin by taking a look at what we'll end up with, and then determine what it will take to get there. First, the standalone code will access its global variables by using the A5 register; there's no way around that. Even if we were to pass the object code through a postcompilation tool that converted all A5 references into A4 references, there's no way we could take care of the cases where the compiler generates code that uses A4 for other purposes. Therefore, this solution still uses A5 for accessing globals.

Second, the globals will be tacked onto the end of the standalone code resource, just as they are with THINK's solution. This means that the globals will be in a known and easily determined location at all times, relieving us from having to rely on the caller to maintain our globals. When doing this, we inherit the problem THINK code has with not being purgeable, but that's a small price to pay for the ease of use we get in return.

Third, the globals will be in expanded format. The approach taken in the Tech Note requires that our standalone code carry around the baggage of the decompression routines, as well as the compressed data, long after they're no longer needed. Using pre-expanded data means a larger on-disk footprint, but again, this is a small price to pay, especially if the in-memory footprint is more of an issue (and it usually is).

Finally, we'll need routines that calculate and set our A5 value when we enter our standalone code, and that restore A5 when we leave. These routines are analogous to the macros THINK uses and to the OpenA5World and CloseA5World routines of the Tech Note solution. Figure 5 shows how our standalone code resource will end up looking, both on disk and in memory.

My system is called StART, for StandAlone RunTime. It consists of two parts: an MPW tool called MakeStandAlone that converts a simple program like the one shown in Figure 3 into a standalone code resource, and a small library file with accompanying header files for Pascal and C.

To show how these pieces work together, let's take a small sample that uses a global variable, and build it using the StART tools. The sample we'll use is the Persist.p

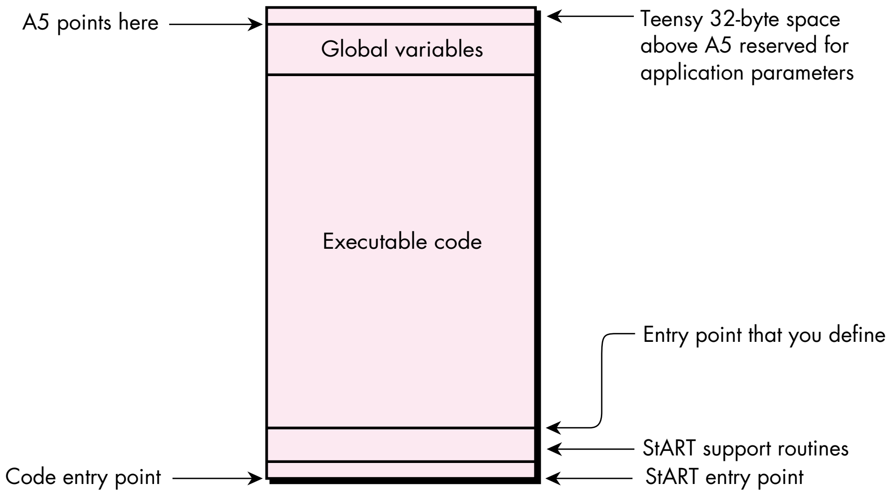


Figure 5
Format of Standalone Code Using StART Techniques

program included in the Tech Note. Following is a version of the file, modified to make calls to the StART library.

```
UNIT Persist;
{ This is a standalone module that maintains a running total of the }
{ squares of the parameters it receives.                            }

INTERFACE
   USES Types, StART;
   FUNCTION Main(parm: LONGINT): LONGINT;
IMPLEMENTATION
   { Define global storage to retain a running total over multiple }
   { calls to the module.                                          }
   VAR
      accumulation:     LONGINT;
   FUNCTION Main(parm: LONGINT): LONGINT;
      VAR
         saved:           SaveA5Rec;
      BEGIN
         UseGlobals(saved);
         accumulation := accumulation + (parm * parm);
         Main := accumulation;
         DoneWithGlobals(saved);
      END;
END.
```

This very simple sample performs the useless function of taking the number you pass it, squaring it, adding the result to a running total, and returning that total. UseGlobals is the StART routine that enables us to access our global variables (in this case, the lone variable named accumulation), returning the value of the caller's A5. After we've performed our mathematical wizardry, we close up shop by calling a second StART routine, DoneWithGlobals, to restore the previous A5 value.

Following is the makefile for Persist.p.

```
Persist       ƒƒ Persist.p.o Persist.make StARTGlue.a.o
   Link  StARTGlue.a.o ∂
         Persist.p.o ∂
         "{Libraries}Runtime.o" ∂
         "{PLibraries}PasLib.o" ∂
         -sn PASLIB=Main ∂
         -o Persist
   MakeStandAlone Persist -restype CUST -resnum 129 -o Persist.rsrc

Persist.p.o    ƒ Persist.p Persist.make
   Pascal Persist.p
```

This makefile contains a couple of interesting things that are worth examining. The first point to note is that we link with a file called StARTGlue.a.o. This file contains a few useful routines, including UseGlobals and DoneWithGlobals. It also contains a special header routine that performs some crucial setup. This setup needs to be performed before any of our custom code can be executed, so StARTGlue.a.o should be the first file in the link list.

The second interesting thing about the makefile is the statement -sn PASLIB=Main. Recall that MakeStandAlone requires a file that contains the resources shown in Figure 3 in order to perform its magic. Specifically, MakeStandAlone demands that there be only three segments with a single entry point each into CODE 1 and CODE 2. However, when we link with PasLib.o, we create a fourth segment called PASLIB. We therefore get rid of this segment by merging it with the rest of our executable code in CODE 1, the Main segment.

After linking and running the resulting file through the MakeStandAlone tool, we're left with a resource containing standalone code that sets up and uses its own set of global variables. Following are highlights from the Persist sample shown above. Some routines have been removed, since we'll be examining them in depth later.

```
Entry
+0000 00000    BRA.S    Entry+$0014
+0002 00002    DC.B     $0000              ; flags
+0004 00004    DC.B     $43555354          ; resource type (CUST)
```

```
+0008 00008    DC.B     $0081                  ; resource ID (129)
+000A 0000A    DC.B     $0000                  ; version
+000C 0000C    DC.B     $00000000              ; refCon
+0010 00010    DC.B     $00000000              ; cached offset to globals
+0014 00014    BRA      MAIN

[ UseGlobals, DoneWithGlobals, GetSAA5, and CalculateOffset removed ]

MAIN                                           ; from Persist.p
+0000 000076   LINK     A6,#$FFF8
+0004 00007A   PEA      -$0008(A6)             ; UseGlobals(save);
+0008 00007E   JSR      UseGlobals
+000C 000082   MOVE.L   $0008(A6),-(A7)        ; parm * parm
+0010 000086   MOVE.L   $0008(A6),-(A7)
+0014 00008A   JSR      %I_MUL4
+0018 00008E   MOVE.L   (A7)+,D0
+001A 000090   ADD.L    D0,-$0004(A5)          ; add to accumulation
+001E 000094   MOVE.L   -$0004(A5),$000C(A6) ; return as function result
+0024 00009A   PEA      -$0008(A6)             ; DoneWithGlobals(save);
+0028 00009E   JSR      DoneWithGlobals
+002C 0000A2   UNLK     A6
+002E 0000A4   MOVE.L   (A7)+,(A7)
+0030 0000A6   RTS

[ %I_MUL4 removed ]

Globals
+0000 000E4    DC.W     $0000, $0000           ; global var accumulation
+0004 000E8    DC.W     $0000, $0000           ; 32 bytes of app parms
+0008 000EC    DC.W     $0000, $0000
+000C 000F0    DC.W     $0000, $0000
+0010 000F4    DC.W     $0000, $0000
+0014 000F8    DC.W     $0000, $0000
+0018 000FC    DC.W     $0000, $0000
+001C 00100    DC.W     $0000, $0000
+0020 00104    DC.W     $0000, $0000
```

Entry, UseGlobals, DoneWithGlobals, GetSAA5, and CalculateOffset are all routines linked in from the StARTGlue.a.o file; MAIN is from the Persist.p source file; and %I_MUL4 is a library routine from PasLib.o. Following these routines are 36 bytes of data. The first 4 bytes are for our global variable, accumulation. The final 32 bytes are the application parameters above A5 that the system occasionally uses.

Let's take a look at the MAIN function, which shows us accessing our global variable. First, we call UseGlobals to determine what A5 should be and to set A5 to that value.

In this case, UseGlobals will set A5 to point to Globals+$0004, placing our single 4-byte global below A5, and the 32 bytes of system data above A5. Next, we push the value we want to square onto the stack twice and call %I_MUL4 to multiply the two 4-byte values.

Finally, we get to the fun part, where we add the result of %I_MUL4 to our global variable. This is done by the instruction at MAIN+$001A: ADD.L D0,-$0004(A5). This instruction says to take the value in register D0 and add it to the number stored four bytes below A5. Because A5 points to Globals+$0004, this instruction adds D0 to the value starting at Globals.

THE MAKESTANDALONE TOOL

The code above was created by the MakeStandAlone tool. Let's look now at the workhorse function of that tool, ConvertAppToStandAloneCode. It's this function that takes an application conforming to the format shown in Figure 3 and converts it to the standalone resource shown in Figure 5.

ConvertAppToStandAloneCode starts by declaring a ton of variables, all of which are actually used. It then opens the file containing the segments shown in Figure 3 by calling OpenResFile on gInputFile, a string variable set up before calling this routine. If we can't open the file, we blow out by calling ErrorExit, a routine that prints the string passed to it and then aborts back to the MPW Shell.

```
PROCEDURE ConvertAppToStandAloneCode;

   VAR
      refNum:           INTEGER;
      code0:            Code0Handle;
      code1:            CodeHandle;
      code2:            CodeHandle;
      sizeOfGlobals:    LONGINT;
      expandedGlobals:  Handle;
      myA5:             LONGINT;
      codeSize:         LONGINT;
      address:          CStrPtr;
      err:              OSErr;
      fndrInfo:         FInfo;
      existingResource: Handle;

   BEGIN
      refNum := OpenResFile(gInputFile);
      IF (refNum = - 1) | (ResError = resFNotFound) THEN
         ErrorExit('Error trying to open the source file.', ResError);
```

Loading the segments. ConvertAppToStandAloneCode then scopes out the contents of the file it has just opened.

The first thing it looks at is CODE 0, which contains the application's jump table. If CODE 0 exists and we can load it, we mark it nonpurgeable and call a utility routine, ValidateCode0, to make sure that CODE 0 contains what we expect. Here's what the code looks like:

```
code0 := Code0Handle(Get1Resource('CODE', 0));
IF (code0 = NIL) | (ResError <> noErr) THEN
   ErrorExit('Couldn't load CODE 0 resource.', ResError);
HNoPurge(Handle(code0));
ValidateCode0(code0);
```

MakeStandAlone requires that the input file conform strictly to the format shown in Figure 3. Among other things, this means that there should be only two entries in the jump table, one for CODE 1 and one for CODE 2. ValidateCode0 checks for this condition and makes a few other sanity checks to make sure that CODE 0 doesn't contain any other information that we'd otherwise have to deal with. If there are any problems, ValidateCode0 calls ErrorExit with an appropriate message. Thus, if ValidateCode0 returns, everything appears to be OK with CODE 0.

At times it might be tricky or impossible to create a CODE 1 resource with only one entry point. In some cases, you can bludgeon your code into a single segment by passing **-sn** to the Link tool, as was done earlier. Unfortunately, this won't always work. For instance, some MPW routines are compiled to require jump table entries. (Examples of such routines are sprintf and its subroutines.) If you try to use any of these routines, you'll get more than one entry point in CODE 1. The only way to avoid this problem is to keep away from library routines that require jump table entries. If you're in doubt, simply attempt to use the routine in question; the compiler, the linker, or MakeStandAlone will tell you if anything is wrong.

ConvertAppToStandAloneCode next checks the remaining resources, CODE 1 and CODE 2. CODE 1 contains the executable code that will make up the bulk of the standalone code resource, and CODE 2 contains the compressed data holding the global variables' initial values, as well as the routines that decompress that data. Each segment is loaded and passed to ValidateCode to make sure that the resource looks OK.

```
code1 := CodeHandle(Get1Resource('CODE', 1));
IF (code1 = NIL) | (ResError <> noErr) THEN
   ErrorExit('Couldn't load CODE 1 resource.', ResError);
HNoPurge(Handle(code1));
ValidateCode(code1, 1, 0);
```

```
code2 := CodeHandle(Get1Resource('CODE', 2));
IF (code2 = NIL) | (ResError <> noErr) THEN
   ErrorExit('Couldn't load CODE 2 resource.', ResError);
HNoPurge(Handle(code2));
ValidateCode(code2, 2, 8);
```

ValidateCode takes a handle to the segment, along with a couple of values used in the sanity check. The first number is actually the resource ID of the segment and is used when reporting any errors. The second value is the jump table offset of the entry point for this segment and is checked against the segment header (see *Inside Macintosh* Volume II, page 61, for a description of this header). Again, if any problems are discovered or any unexpected values encountered (such as more than one entry point per segment), ValidateCode aborts by calling ErrorExit.

Converting to a standalone resource. Once the three segments have been loaded into memory and validated, we're ready to convert these resources into a single standalone resource. We begin by decompressing the data that represents the preinitialized values for our global data. The first part of accomplishing this is getting a temporary buffer to hold the expanded values. We find the size of this buffer by looking at the belowA5 field in CODE 0. We then create a buffer this size by calling NewHandle.

```
sizeOfGlobals := code0^^.belowA5;
expandedGlobals := NewHandle(sizeOfGlobals);
IF expandedGlobals = NIL THEN
   ErrorExit('Couldn't allocate memory to expand A5 data.', MemError);
```

We next perform the magic that expands the global variables into the buffer. CODE 2 contains the decompression routines, so all we do is call them. The function that performs this decompression is called _DATAINIT, which our validation routines have already confirmed is the entry point to CODE 2. _DATAINIT needs to have A5 already pointing to the top of the globals area, which in our case is the end of the handle we just created. After calling SetA5 to do this, we use CallProcPtr, a little inline assembly routine, to call _DATAINIT in CODE 2. _DATAINIT fills in our handle with the initial values for our global variables and then kindly returns to us. We quickly restore the previous value of A5 so that we can access our own global variables again, and then prepare to finish with the input file. We'll need CODE 1 later, so we detach it from the input file, and then close the input file.

```
myA5 := SetA5(ord4(expandedGlobals^) + sizeOfGlobals);
CallProcPtr(ProcPtr(ord4(code2^) + SizeOf(CodeRecord)));
myA5 := SetA5(myA5);
DetachResource(Handle(code1));
CloseResFile(refNum);
```

At this point, we're done with the input file, and we have in our possession two handles. The code1 handle contains the executable code for the standalone resource, and the expandedGlobals handle contains the global data. Our task at this point is to combine these two pieces of data.

We start by getting the size of the actual object code in CODE 1. This is the size of the entire handle, less the size of the CODE resource header. The handle is then grown large enough to hold the object code, the global data, and the 32 bytes of application parameters. If we can't grow the handle, we exit. Game over.

```
codeSize := GetHandleSize(Handle(code1)) - SizeOf(CodeRecord);
SetHandleSize(Handle(code1), codeSize + sizeOfGlobals + kAppParmsSize);
IF MemError <> noErr THEN
   ErrorExit('Couldn't expand CODE 1 handle.', MemError);
```

Once the handle containing the code is large enough, we call BlockMove twice to put everything in place. The first call to BlockMove moves the object code down in the handle, effectively removing the segment header. This header is useful only for segments and jump table patching; we don't need it for our standalone resource. The second call to BlockMove copies the global data stored in expandedGlobals to the end of the handle holding the object code. We finish up by calling FillChar, a built-in Pascal routine, to clear out the 32 bytes of application parameters.

```
BlockMove(Ptr(ord4(code1^) + SizeOf(CodeRecord)), Ptr(code1^), codeSize);
BlockMove(expandedGlobals^, Ptr(ord4(code1^) + codeSize), sizeOfGlobals);
address := CStrPtr(ord4(code1^) + codeSize + sizeOfGlobals);
FillChar(address^, 32, CHAR(0));
```

Filling out the header. Our standalone code resource is now almost complete. All that remains is to fill out the fields of the standard header that seems to begin most standalone code resources.

The header consists of a word for a set of flags, the type and ID of the resource, and a word for a version number. These fields were written to our original CODE 1 when we linked with StARTGlue.a.o, but they were uninitialized. We take the opportunity here to fill in these fields.

As an additional goodie, our standard header contains a 4-byte refCon that can be used for anything the standalone code wants (for example, holding some data that the calling application can access).

Once the global data has been appended to the object code handle, we no longer need the expandedGlobals handle, so we dispose of it and prepare to write out our *objet d'art*.

```
WITH StdHeaderHandle(code1)^^ DO BEGIN
   flags := gHdrFlags;
   itsType := gResType;
   itsID := gResID;
   version := gHdrVersion;
   refCon := 0;
END;

DisposeHandle(expandedGlobals);
```

Writing the standalone resource. The first step to writing out our standalone code resource is to open the file that will hold it. We do this by calling OpenResFile. If OpenResFile reports failure, it's probably because the file doesn't exist. Therefore, we try to create the file by calling CreateResFile. If that succeeds, we set the Finder information of the output file so that we can easily open it with ResEdit, and then attempt to open the file again. If that second attempt fails, we give up by calling ErrorExit.

```
refNum := OpenResFile(gOutputFile);
IF (refNum = - 1) | (ResError = resFNotFound) THEN BEGIN
   CreateResFile(gOutputFile);
   IF (ResError <> noErr) THEN
      ErrorExit('Error trying to create the output file.', ResError);

   err := GetFInfo(gOutputFile, 0, fndrInfo);
   IF err <> noErr THEN
      ErrorExit('Error getting finder information.', err);

   fndrInfo.fdType := 'rsrc';
   fndrInfo.fdCreator := 'RSED';
   err := SetFInfo(gOutputFile, 0, fndrInfo);
   IF err <> noErr THEN
      ErrorExit('Error setting finder information.', err);

   refNum := OpenResFile(gOutputFile);
   IF (refNum = - 1) | (ResError = resFNotFound) THEN
      ErrorExit('Error trying to open the output file.', ResError);
END
```

If our first call to OpenResFile succeeded (skipping to the ELSE clause shown below), the file already exists and may need to be cleaned up a little. If the output file already contains a resource with the same type and ID of the resource we want to write, we need to get rid of it. Calls to RmveResource and DisposeHandle accomplish that grisly task.

```
ELSE BEGIN
   SetResLoad(FALSE);
   existingResource := Get1Resource(gResType, gResID);
   SetResLoad(TRUE);

   IF existingResource <> NIL THEN BEGIN
      RmveResource(existingResource);
      DisposeHandle(existingResource);
   END;
END;
```

At this point, we have a handle that needs to be added to a file as a resource, and an open file waiting for it. Three quick calls to the AddResource, WriteResource, and SetResAttrs routines take care of the rest of our duties, and the standalone code resource is written to the designated file. We then close the file and leave ConvertAppToStandAloneCode with the knowledge of a job well done.

```
   AddResource(Handle(code1), gResType, gResID, gResName);
   IF ResError <> noErr THEN
      ErrorExit('Error adding the standalone resource.', ResError);

   WriteResource(Handle(code1));
   IF ResError <> noErr THEN
      ErrorExit('Error writing the standalone resource.', ResError);

   SetResAttrs(Handle(code1), gResFlags);
   IF ResError <> noErr THEN
      ErrorExit('Error setting the resource attributes.', ResError);

   CloseResFile(refNum);
END;
```

UP CLOSE AND PERSONAL WITH STARTGLUE.A.O

Converting our application into a standalone code resource is only part of the process. The other part involves the routines that allow our code to execute on its own. These routines preserve the A5 world of the host application, set up the standalone code's A5 world, and restore the host application's A5 world when the standalone code is finished.

These routines are provided by StARTGlue.a.o. StARTGlue.a.o includes four client (external) routines (UseGlobals, CopyHostQD, DoneWithGlobals, and GetSAA5), an internal routine (CalculateOffset), and a block of public and private data. Because of this embedded block of data, the library is written in assembly language. Let's take a look at the source file, StARTGlue.a.

```
                  CASE        OFF

                  INCLUDE     'Traps.a'
                  INCLUDE     'QuickEqu.a'
                  INCLUDE     'SysEqu.a'

FirstByte         MAIN
                  IMPORT      Main, _DATAINIT
                  ENTRY       gGlobalsOffset
                  bra.s       Island

                  dc.w        0                 ; flags
                  dc.l        0                 ; resType
                  dc.w        0                 ; ID
                  dc.w        0                 ; version
                  dc.l        0                 ; refCon

gGlobalsOffset    dc.l        0                 ; offset to globals
```

By convention, standalone code resources start with a standard header having the format shown in Table 2.

Table 2
Standard Header for Standalone Code Resources

Field	Size	Contents
entry	2 bytes	Branch instruction to first byte of executable code.
flags	2 bytes	User-defined flags. You can set and define this field any way you want.
resType	4 bytes	Resource type.
resID	2 bytes	Resource ID.
version	2 bytes	Version number. The values for this field are unregulated, but usually follow the same format as the version numbers in 'vers' resources.
refCon	4 bytes	User-defined reference constant. Use this field for anything you want, including communicating with the host.

Nothing requires standalone code to include this header. However, it's nice to follow convention, and including the resource type and ID makes identifying blocks in the heap easier.

When you compile and link with StARTGlue.a.o, these fields are empty (set to zero). However, the MakeStandAlone tool automatically fills in these fields based on command-line options when it converts your code.

StARTGlue.a.o's entry point branches to the following code, which then branches to a function called Main. The reason for this double jump is to maintain the standard header for a standalone code resource. The first two bytes are used to jump to the code's entry point. However, we can jump only 128 bytes with the 68000's 2-byte relative branch instruction. If Main happens to be further than 128 bytes from the start of the code resource, we would need to use the 4-byte branch instruction. To provide for this contingency, we have our 2-byte branch instruction jump to the 4-byte branch instruction, which can then jump to anywhere that it wants with impunity.

```
Island
                  bra        Main
                  lea        _DATAINIT,A0      ; dummy line to reference
                                               ;   _DATAINIT
```

The LEA instruction that follows the branch is a dummy statement. Its sole purpose is to trick the linker into including _DATAINIT, the routine that the MakeStandAlone tool calls to decompress the global data. Because the LEA instruction immediately follows an unconditional branch, and because it doesn't have a label that can be jumped to, it's never actually executed.

UseGlobals. The UseGlobals function is used to set up the standalone code's A5 world. An example of this is shown earlier in the Persist program.

UseGlobals performs three functions:

- It sets the A5 register and the low-memory location CurrentA5 to the correct value for the standalone code. It determines the standalone code's A5 value by calling the GetSAA5 function, described later.
- It copies the host application's QuickDraw globals pointer to the standalone code's QuickDraw globals pointer (this pointer is the 4-byte value to which A5 normally points). By copying this pointer, the standalone code can call Toolbox routines knowing that A5 references a valid set of QuickDraw globals.
- It returns the host application's A5 and CurrentA5 values so that they can later be restored.

```
;
; PROCEDURE UseGlobals(VAR save: SavedA5Rec);
; { Balance with DoneWithGlobals. }
;
UseGlobals        PROC       EXPORT
                  IMPORT     GetSAA5
                  move.l      4(sp),A0          ; get ptr to save record
                  move.l      A5,(A0)           ; save A5
                  move.l      CurrentA5,4(A0)   ; save low-memory value
                  clr.l       -(sp)             ; make room for function
                                                ;   result
                  bsr.s       GetSAA5           ; get our own A5
                  move.l      (sp)+,A5          ; make it real
                  move.l      A5,CurrentA5      ; make it really real
                  move.l      4(sp),A0          ; get ptr to save record
                  move.l      (A0),A0           ; get host's A5
                  move.l      (A0),(A5)         ; copy his QD globals ptr
                  move.l      (sp)+,(sp)        ; remove parameters
                  rts                           ; return to caller
```

CopyHostQD. The CopyHostQD routine is an optional utility routine. You don't need to call it unless you have to ensure that the host's QuickDraw globals remain undisturbed. By default, your standalone code shares the same set of QuickDraw globals as the host application. However, if you have unusual requirements, you may need to establish your own set of QuickDraw globals.

A simple way to set up your own QuickDraw globals would be to call InitGraf(@thePort) after you called UseGlobals. This would create a valid set of QuickDraw globals. However, some standalone code resources initially need to work with information provided by the host application. For instance, a custom MDEF normally draws in the currently set port. To inherit such information, you can call CopyHostQD just after you call UseGlobals.

```
;
; PROCEDURE CopyHostQD(thePort: Ptr; oldA5: Ptr);
;     { Balance with DoneWithGlobals. }
;  assumes that A5 has already been set up to our globals
;
CopyHostQD        PROC        EXPORT

returnAddress     EQU         0
oldA5             EQU         returnAddress+4
thePortPtr        EQU         oldA5+4
parameterSize     EQU         thePortPtr-oldA5+4

                  move.l      oldA5(sp),A0      ; get oldA5
                  move.l      (A0),(A5)         ; make (A5) point to
                                                ;   thePort

                  move.l      (A0),A0           ; get host's thePort
                                                ;   pointer
                  move.l      thePortPtr(sp),A1 ; get our thePort pointer
                  move.l      #grafSize,D0      ; copy whole grafPort
                  move.l      D0,D1             ; since the pointers
                  subq.l      #4,D1             ;   point near the end of
                  sub.l       D1,A0             ;   the QD globals, move
                  sub.l       D1,A2             ;   them down to point
                                                ;   to the beginning
                  _BlockMove

                  move.l      (sp)+,A0          ; pop return address
                  add         #parameterSize,sp ; pop parameters
                  jmp         (A0)              ; return to caller
```

DoneWithGlobals. The DoneWithGlobals routine reverses the effects of UseGlobals. It simply restores the values of the A5 register and low-memory global CurrentA5 to the values saved by UseGlobals.

```
;
; PROCEDURE DoneWithGlobals(restore: SaveA5Rec);
;
DoneWithGlobals   PROC        EXPORT

                  move.l      (sp)+,A0          ; pull off return address
                  move.l      (sp)+,A1          ; address of record
                                                ;   holding info
                  move.l      (A1),A5           ; first restore A5
                  move.l      4(A1),CurrentA5   ; then restore low-memory
                                                ;   value
                  jmp         (A0)              ; return to caller
```

GetSAA5. You probably won't need to call GetSAA5. This function is called by UseGlobals to return the value that's used to refer to the standalone code's A5 world. The first time this function is called, this value needs to be calculated. After that, the offset from the beginning of the code to the global data is cached and is used in subsequent calls to GetSAA5. Once the offset has been determined, it's added to the address of the start of the standalone code and returned to the caller.

```
;
; FUNCTION GetSAA5: LONGINT;
;
GetSAA5           PROC        EXPORT
                  IMPORT      CalculateOffset

                  move.l      gGlobalsOffset,D0 ; have we done this
                                                ;   before?
                  bne.s       @1                ; yes, so use cached
                                                ;   value
                  bsr.s       CalculateOffset   ; nope, so calculate it
@1
                  lea         FirstByte,A0      ; get base address
                  add.l       A0,D0             ; add offset to top of
                                                ;   globals
                  move.l      D0,4(sp)          ; set function result

                  rts                           ; return to caller
```

CalculateOffset. CalculateOffset determines the offset from the beginning of the code resource to the location that A5 should point to. We see from Figure 5 that A5 should point to the location 32 bytes before the end of the resource. Therefore, we get a handle to the code resource, get the code resource's size, subtract 32 from it, and return the result as the needed offset.

```
CalculateOffset   PROC

                  lea         FirstByte,A0      ; get pointer to us
                  _RecoverHandle                ; get handle to us
                  _GetHandleSize                ; find our size (= offset
                                                ;   to end of globals)
                  sub.l       #32,D0            ; account for 32 bytes of
                                                ;   appParms
                  lea         gGlobalsOffset,a0 ; get address to save
                                                ;   result
                  move.l      D0,(A0)           ; save this offset for
                                                ;   later
                  rts
```

SUMMARY OF THE THREE SOLUTIONS

This article has explored three ways to access global variables in standalone code: the THINK method, the OpenA5World method, and the StART method.

The THINK method uses the A4 register to access the global variables. The A4 register is managed by the RememberA0, SetUpA4, and RestoreA4 functions. The advantages of the THINK method are as follows:

- The host's A5 register is untouched.
- The storage for globals is coupled with the storage for the code itself, meaning that no additional storage needs to be allocated or disposed of.

The disadvantages of the THINK method are:

- The A4 register cannot be used for code optimization.
- Standalone code resources cannot be marked purgeable without the risk of losing any values stored in global variables.
- Unless you use the multisegmented standalone code features of the THINK environments, you're limited to a combined total of 32K of code and data.
- The global data is stored in an uncompressed format on disk.

Because MPW doesn't provide the compiler support that THINK does, the approach described in the Tech Note reuses register A5 to access global variables. Support is provided by the functions MakeA5World, SetA5World, RestoreA5World, DisposeA5World, OpenA5World, and CloseA5World. The advantages of this method are as follows:

- It has a compact on-disk format (global data is compressed).
- A4 is free for code optimization.
- The code resource can be marked purgeable.
- You can access 32K of code and 32K of data.

The disadvantages of the Tech Note method are:

- It requires support from the host application for persistence of globals.
- Care must be taken to restore the host's A5 when control is returned to the host (which can include callbacks, a la HyperCard).

The StART solution tries to incorporate the best of both worlds. StART's use of the A5 register is managed by calls to UseGlobals, DoneWithGlobals, and (optionally) CopyHostQD. Its advantages are as follows:

- A4 is free for code optimization.
- You can access 32K of code and 32K of data.
- The storage for globals is coupled with the storage for the code itself, meaning that no additional storage needs to be allocated or disposed of.

The disadvantages it doesn't address are:

- Care must be taken to restore the host's A5 when control is returned to the host (which can include callbacks).

- Standalone code resources cannot be marked purgeable without the risk of losing any values stored in global variables.
- The global data is stored in an uncompressed format on disk.

There's one major limitation that none of these techniques address. Neither MPW nor THINK can handle certain kinds of global variables — ones that get preinitialized to some absolute address — in standalone code. For instance, consider the following C source:

```
char *myStrings[] = {
   "Macintosh",
   "Programming",
   "Secrets",
   "2nd Edition"
};
```

This declares an array of pointers to the four given strings. When this definition appears in source code in a THINK C project, the compiler will tell you that this sort of initialization is illegal in standalone code. However, MPW's compilers aren't as integrated into the build process as THINK's are, and they don't know to give you a similar warning. Thus, we can compile an array like the one just shown without an error. When the MakeStandAlone tool is later executed, it will dutifully initialize the array with pointers to the given strings. However, these pointers are in the form of absolute memory locations, which are valid only at the time the globals are expanded. When it's time to execute the standalone code, it's almost certain that the strings won't be loaded into the same place they were in when the globals were expanded, making the pointers in our array invalid.

All you can do to avoid this problem is make sure that you don't have any global variables that are preinitialized to the addresses of other objects (such as strings, functions, and other variables). Without knowing the format of the compressed global data that _DATAINIT expands, it isn't possible to program the MakeStandAlone tool to look for the problem globals.

WHERE TO GO FROM HERE

This article just scratches the surface of what can be done with MPW. It gives a little behind-the-scenes information and describes how to take advantage of that information with a custom tool. The intrepid explorer may want to apply what's learned here to some other topics.

32-BIT EVERYTHING

With MPW 3.2, Apple has eliminated most of the traditional 32K barriers imposed by 16-bit fields. By expanding fields in the jump table to 32 bits, replacing the

Segment Loader, patching object code with absolute addresses, and providing user-callable runtime routines, MPW allows you to create code and data blocks of practically any size. It may be interesting to explore the new formats and data structures used with 32-bit everything to see how you can use them in the same way we used the old 16-bit information.

MERGING START TECHNIQUES WITH THOSE OF THE TECH NOTE

The StART method uses a bit of assembly language to provide some runtime support for standalone code. Specifically, it maintains a reference to the code's global variables in a local data field. This same technique could be used to partially remove the dependency of code created with the Tech Note method on the host application.

JUMP TABLE

We've fully explored the area below A5, but only a small part of the area above A5. We've looked at the globals area below A5 and the application parameters area above A5, but the majority of the "above A5 world" is normally occupied by a jump table that supports multisegmented applications. With a little more work and runtime support, it may be possible to write multisegmented standalone code in MPW.

Multisegmented standalone code offers more benefits than simply allowing you to write huge chunks of standalone code. Programmers using Object Pascal and readers of the Macintosh Technical Note "Inside Object Pascal" (formerly #239) know that polymorphism requires the use of a jump table. By implementing support for a jump table in standalone code, it should be possible to write standalone code with Object Pascal or C++'s PascalObjects. C++ programmers writing native C++ classes or classes based on HandleObject should refer to Patrick Beard's article, "Polymorphic Code Resources," in *develop* Issue 4.

THANKS DEPARTMENT

This article would not have existed if not for the help and inspiration of the following individuals and nonindividuals:

- The creators of the A4 method used in the THINK products for showing that globals could be used in standalone code
- The authors of the BuildDCMD tool for MacsBug, a tool that proved that applications conforming to a certain guideline could be converted to standalone code
- Larry Rosenstein, who, thanks to file sharing, unknowingly provided the source code shell for the MakeStandAlone tool (all the stuff that deals with error handling and command-line parsing)

THANKS TO OUR TECHNICAL REVIEWERS
C. K. Haun, Pete Helme, Craig Prouse•

DAVE JOHNSON

THE VETERAN NEOPHYTE

DIGITAL ZOOLOGY

4 A.M. Friday still feels like Thursday. Five hours remain until the contest. Bean dip slowly dries around the rim of a jar, turning a darker, almost translucent brown. This corner of the table, the one nearest the center of the room, is littered with the strange and particular combination of plastic, paper, metal, glass, and organic debris that typifies the remains of junk food. The room, a large but nondescript meeting room with beige-painted cinder block walls, is bathed in fluorescent light, 60-cycle radiation painting the few remaining occupants a lovely whitish green.

A few of them still hunch over keyboards, pecking feverishly, squeezing the last few desperate instructions into their robots. Others sprawl on the floor around the test course, watching carefully and hopefully as their fragile creations, their little Lego and wire and motor golems, their tiny mind children, haltingly — but autonomously — negotiate their way toward the goal. The expressions on their faces are variously rapt, worried, and proud.

The scene is the early morning of the last day of Artificial Life III, a week-long scientific hoe-down that took place last June in Santa Fe. The hardy hackers in the cluttered room at the back of the building are entrants in a robot-building contest that will be run as part of the "Artificial 4H Show" beginning at 9 A.M. Their robot creatures run the gamut from the eminently practical to the practically insane.

The insane ones, of course, are by far the more interesting. One, appropriately named Rob Quixote, has only a single wheel, and therefore must steer by rotating an oversized horizontal windmill-like contraption fastened to its head, effectively pushing against the air to turn itself. Another moves by a sort of spastic lurching; throwing its entire front section forward, it gains an awkward quarter inch, then gathers up its hindquarters for another throw. This one is so inefficient that it requires twice the usual number of batteries, and uses them up in a single run. Amazingly enough, though, it successfully traverses the course, albeit slowly and with much ineffectual thrashing.

"Artificial life," as a named discipline, appeared on the scientific scene relatively recently. The first conference happened in the fall of 1987, and gave joyous birth to this new field of scientific inquiry, or rather this new and rich confluence of many different fields. Scientists who had been working in isolation suddenly discovered others pursuing similar lines of investigation, and the meeting of minds was electric.

Artificial life is an attempt to create and study artificial systems — that is, systems created by humans — that mimic processes or exhibit behaviors usually associated only with living systems. Predictably, the primary medium that these systems are created on (in?) is computers; this is a field that depends heavily on technology to get its work done (they're doomed if electricity ever becomes unavailable). Also predictably, a large proportion of its devotees are biologists, especially theoretical biologists.

Why would biologists want to study artificial life? Don't they already have their hands full trying to figure out the real thing? Well, for one thing, there are a lot of experiments biologists would love to do that they simply can't: nature doesn't come with convenient levers and knobs, and you can never roll back time and try something over again. So if biologists can develop good models of biological phenomena, they can implement them on computers and run clean and tidy experiments that are repeatable, detailed, controlled, and manipulable down to the last detail. This is a far

DAVE JOHNSON's mother recently moved across the country, and sent him a total of eight large cardboard boxes crammed with junk spanning his entire life that she didn't want cluttering her garage any more. Among his old school stuff was a report card from second grade that included a couple of N's, meaning "needs improvement." The N's were in the categories of "Is Prompt" and "Works Steadily." Here's a quote from his teacher, Mrs. Doris Short, that accompanied the report: "We've talked about being prompt, but it's always 'I'll finish tomorrow.'" This is strong evidence for the claim that personality is established early in life, and never changes.•

cry from the messy, inexact, unrepeatable real world, and for some biologists would be tantamount to scientific nirvana.

But there's another, larger reason for biologists to study artificial life. In the words of Chris Langton, self-described "midwife" of artificial life (he organized the first conferences and named the field), "Such systems can help us expand our understanding of life as it *could* be. By allowing us to view the life that has evolved here on Earth in the larger context of *possible* life, we may begin to derive a truly general theoretical biology capable of making universal statements about life wherever it may be found and whatever it may be made of."

I like it.

When I read this I was hooked. Visions of bizarre, unknowable alien intelligences and strange, seething soups that cling and quiver and creep around filled my head. And here are real scientists hanging around seriously discussing it! This is some serious fun! And lots of different kinds of scientists are paying attention; biologists, mathematicians, physicists, chemists, robotocists, and computerists are all well represented at the conferences, with a sprinkling of philosophers, anthropologists, economists, and others. The gee-whiz factor hooked me, but the interdisciplinary thrust of artificial life reeled me in.

(In conversation people say "*a*-life." I've seen it written as Alife, A-life, alife, and a-life. I wanted to use alife, but people tended to pronounce it like "get a life," so I'll use a-life instead.)

Another appeal for me is the tacit approval of the "build it first, then study it" approach in a-life. This method of building things and learning things (stumbling around, really, but *intelligent* stumbling, *directed* stumbling) has always been my particular forte. The premise is that we don't need to completely understand something before we can build it or build a model of it, and that it's very often more instructive to get a crude version up and working immediately than to try to refine the thing completely before trying it out. By fumbling around and building things blindly, we can often learn a lot by virtue of the happy accidents that inevitably occur. And it's *tons* more fun that way.

There were far too many interesting things at the conference to describe them all here. Instead I want to tell you about one particular talk that caused me to have a powerful "Aha!" experience (and I *live* for "Aha!" experiences). If you know something about evolution already, the following may not be news to you, but presumably most computer programmers don't study biology.

The talk dealt with Lamarckian evolution. Lamarck was a contemporary of Darwin who postulated that the things experienced by an organism during its lifetime could affect the traits handed down to the next generation. As an example, a Lamarckian might believe that proto-giraffes had to stretch their necks up to reach the leaves at the tops of the trees, and because of all the stretching, their descendants were born with longer necks. Unfortunately for Lamarck and his followers, this is rubbish.

It turns out that as far as biological evolution is concerned, Lamarckism is nonexistent: there was no such thing at work in the development of life on Earth. So my curiosity was piqued when I saw the title of this talk by David Ackley and Michael Littman: "A Case for Distributed Lamarckian Evolution." What, were they crazy? Talking Lamarck to all these modern scientists? (At the previous conference, Ackley had one of the few really amusing presentations, so of course I would have gone no matter what the topic, but this one looked particularly juicy.)

Ackley and Littman weren't trying to convince people that Lamarckian evolution had anything to do with life on earth. What they did instead was compare the efficiencies of the two types of evolution. (They created a simple evolution simulation, and then compared Darwinian and Lamarckian evolution in their abilities to find a solution to a particular problem.) Hey, this is after all *artificial* life, so if Lamarckian evolution works better, we can use it, right?

What they found was that when Lamarckian evolution was allowed to enter the picture — when the things learned in one generation were at least partially passed on to the next — the system was much, much better at solving the given problem. It consistently found better solutions faster in every single case they tried. This of course makes some intuitive sense. Rather than waiting for genetic shuffling to find a solution to the problem, the prior generation can point the current one in the right direction. So Lamarckian evolution is pretty much a great thing, evolutionarily speaking, because it gets you a lot further and it gets you there a lot faster. (Where it is exactly that you're going is a question for the philosophers; for the moment, let's just blithely assume that we really *do* want to get there.) Their point was that as simulation builders we should think about using Lamarckian inheritance in our simulations, because it works so well. But this point reinforced something else that had been rolling around in my head.

There's an evolutionary premise that I initially learned about through reading an article by a robotocist named Hans Moravec in the first Artificial Life proceedings. I learned more about it in Richard Dawkins's book *The Blind Watchmaker* and in a fascinating book called *Seven Clues to the Origin of Life* by a Glasgow chemist named Graham Cairns-Smith. This particular concept is called "genetic takeover."

According to this idea, one substance can gradually replace another as the carrier of genetic information. Cairns-Smith postulates that life began with replicating inorganic crystals — clays, as a matter of fact — and that a genetic takeover gradually occurred, with proteins and nucleic acids gaining in dominance until finally the original materials were no longer needed. Dawkins and Moravec (and many others) think that a genetic takeover is occurring now, with human culture taking over from nucleic acids as the evolving entity, though they differ in their candidates for the new "gene-equivalent."

Dawkins likes to speak about the "meme," a very useful term first coined in his book *The Selfish Gene*. A meme is an idea, really, or a piece of information. It is immaterial, and requires a material substrate of brains, books, computers, or other media to exist. But given that substrate, the parallels with genes are very good. Just like genes, memes replicate (we tell each other good ideas, or write them down for others), memes mutate (we don't always get it right in the telling), memes mate (ideas in combination often give birth to new ones), and memes compete for survival ("good" ideas stick around a long, long time, but "bad" ones die by not being passed on to anyone: mindshare is their means of existence).

Moravec, on the other hand, seems to be more interested in the evolution of machines, and speculates convincingly and entertainingly that our machines, our artifacts, will eventually become the dominant evolving entities on Earth. Science fiction, or science fact? I don't know — there are compelling arguments both ways — but in either case it makes for very good reading.

In any case, they think that perhaps here on Earth biological evolution is thoroughly obsolete, and almost despite myself I have to agree. Sure, it's still operating, but the evolution of human bodies has been completely outstripped by the evolution of human culture. Bodies evolve at an extremely slow pace, but culture evolves incredibly fast, and humans are having such a profound impact on the Earth that biology simply can't keep up. Look at the changes on Earth in the last millennium. Most of the species alive a thousand years ago have remained physically about the same, yet there's no question that the Earth has undergone a radical transformation, and primarily at the hands of humans, as a by-product of their culture. (You might hesitate to call the rampant, wanton destruction and boundless consumption of resources that Earth has suffered at the hands of humans "evolution," but remember that the word "evolution" does *not* necessarily imply improvement.) But why is it going so fast? How come humans do this and other species don't?

One of the primary distinctions between human beings and their close animal relatives is language. Humans can communicate with abstract symbols, and their communications can be "fossilized" in time (that is, written down for later). Here comes the "Aha!" we've all been waiting for: this ability allows humans to engage in a form of Lamarckian evolution! The things we learn in our lifetimes *can* be passed on to the next generation, though in a filtered sort of way. We can't change the way our offspring are built, but we *can* change their behavior (teenagers notwithstanding). Other species do this to some extent, but humans are the unquestioned champs at shaping their offspring.

As you can see, a-life — just like life itself — is rife with philosophical conundrums and radical, thought-provoking concepts, and that's much of the reason I stay interested. But probably the biggest reason of all that I like a-life is hard to express, except by analogy: I get the same feeling peering through a glass screen into a computer world full of digital critters that I do peering through the bars of a cage at the zoo. The xenophile in me wants to see all the forms that life can take, and get to know the minds of every other being. I want to puzzle out the motivations behind a critter's behavior, and I love that shock of recognition I experience every time I look into an animal's eyes — even the ones that are so alien, like birds and reptiles and fish. Again, it's this feeling that there are universal properties of life waiting to be discovered, properties that apply not only to life as it has evolved on Earth but to all *possible* life, including the digital variety.

Are any of these a-life explorations really alive? That's an energetic and ongoing debate among a-lifers, of course, and the answer ultimately depends on the definition you pick for the word "life." Rather than arguing whether metabolism is more necessary to life than reproduction, though, I like to duck the definition issue. I don't really care too much whether we *call* them alive, I want to see if people react to them *as if* they're alive. I want to see that shock of recognition occur when people and digital organisms collide. (What if "they" recognize "us"?!) It's sort of the Turing Test approach for life: if it seems alive — if people can't tell that it's *not* alive — then no matter what we call it, people will treat it as if it's alive. *That* I'd like to see.

RECOMMENDED READING

- *Artificial Life* by Steven Levy (Pantheon Books, 1992).
- *The Blind Watchmaker* by Richard Dawkins (W. W. Norton & Company, 1987).
- *The Selfish Gene* by Richard Dawkins (Oxford University Press, 1976).
- *Seven Clues to the Origin of Life* by A. G. Cairns-Smith (Cambridge University Press, 1985).
- *ZOTZ!* by Walter Karig (Rinehart & Company, Inc., 1947).

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

MACINTOSH Q & A

Q *Here's a tidbit I stumbled across in Inside Macintosh Volume VI, page 3-10: the four Dialog Manager procedures CouldDialog, CouldAlert, FreeDialog, and FreeAlert are no longer supported. I use CouldDialog, and I happened to notice that it didn't work right when I tested it under System 7, but I reported it as a bug. Now you tell us that it's not guaranteed to work in System 7. I can't recall a trap ever becoming suddenly unsupported like this. What's the story?*

A The system software engineers felt that CouldDialog, CouldAlert, FreeDialog, and FreeAlert didn't do much good under System 6, since the Could calls never completely guaranteed that all dialog items were loaded in. These calls also caused problems in the beta versions of System 7. Relatively little software uses those traps anymore; like many things in *Inside Macintosh* Volume I, they're relics of the days when Macintosh programmers had to deal with desk accessory and floppy disk support issues. So these calls were simply patched out. In the final System 7, the traps return without doing anything.

Q *I can't get the black-and-white version of my lasso-type tool to work correctly with CalcMask and CopyMask. With CalcCMask it seems to work fine. What could I be doing wrong?*

A CalcMask and CalcCMask are similar in that they both generate a one-bit mask given a source bitmap. With CalcCMask, though, a pixMap can be used in place of the source bitmap; the seedRGB determines which color sets the bits in the mask image. An easy mistake to make is to forget that CalcCMask expects a pointer to a BitMap data structure while CalcMask expects a pointer to the actual bit image. And unlike CalcCMask, which uses bounding rectangles for the image's dimensions, CalcMask uses the bitmap's rowBytes and pixel image offsets to determine the bounding Rects for the image. A typical call to these routines is as follows:

```
BitMap   source, mask;
CalcMask (source.baseAddr, mask.baseAddr, source.rowBytes,
   mask.rowBytes, source.bounds.bottom-source.bounds.top,
   source.rowBytes>>1);
CalcCMask (&source, &mask, &(*source).bounds, &(*mask).bounds,
   &seedRGB, nil, 0);
```

One last thing to note when using CalcMask is that the width of the image is in words and not bytes. To learn more about these routines, see page 24 of *Inside Macintosh* Volume IV and page 72 of *Inside Macintosh* Volume V. Also, the *Developer CD Series* disc contains a sample, CalcCMask&CalcMask, that shows how to use these routines.

Kudos to our readers who care enough to ask us terrific and well thought-out questions. The answers are supplied by our teams of technical gurus; our thanks to all. Special thanks to Chris Berarducci, Tim Dierks, Steve Falkenburg, Marcie "M. G." Griffin, Charles Grosjean, Bill Guschwan, C. K. Haun, Dave Hersey, Dennis Hescox, Rich Kubota, Scott Kuechle, Edgar Lee, Jim Luther, Joseph Maurer, Kevin Mellander, Guillermo Ortiz, Dave Radcliffe, Greg Robbins, Kent Sandvik, Eric Soldan, Brigham Stevens, Dan Strnad, Forrest Tanaka, and John Wang for the material in this Q & A column.•

Q *How do I update the color table of my off-screen graphics world without destroying the picture?*

A The recommended approach for changing the color table of an existing GWorld involves calling UpdateGWorld, passing either clipPix or stretchPix for gWorldFlags. When passed either of these constants, QuickDraw knows to update the pixels of the pixMap image. Even though the actual image isn't changed, the flags are still needed to remap the pixels to their new colors.

Q *Are there any C++ or C compilation flags that will optimize performance of the Macintosh Quadra computers? Even when I use the "-NeedsMC68030" flag in MacApp, an investigation of the MABuild source files reveals that it sets compiler flags only for the 68020 optimization. If Quadra-specific compilation flags don't exist, do you have any Quadra performance optimization suggestions?*

A The current MPW compilers don't have a 68040 performance optimization flag, though Apple's future compilers will optimize code for the best possible '040 performance. In the meantime, here are some tips on '040 performance tuning:

- Cache management for the '040 can give you the biggest performance boost. Keep program loops inside the cache space, and flush the cache as seldom as possible. In most cases you'll have small loops inside the 4K instruction cache.
- You might get better performance by not calling BlockMove, because the system flushes the cache when you call it in case you're moving code. If you're moving data, the cache doesn't need to be flushed, but the system can't tell from the BlockMove call whether you're moving code or data. Testing will help you determine whether you should call BlockMove or write your own transfer routine. The new MOVE16 opcode is used by the BlockMove trap when the system is running on an '040 processor, but because of problems with this opcode in early '040 processors, it requires special handling. For details, see the Macintosh Technical Note "Cache As Cache Can" (formerly #261).
- Transcendental functions aren't implemented in the 68040 hardware as they are in the 68881 chip used with the 68020 and 68030. Consequently, the functions are emulated in software, resulting in slower performance. If you suspect that your floating point performance is less than optimal, consider modifying your code to use functions supported by the internal '040 FPU. See the Macintosh Technical Note "FPU Operations on Macintosh Quadra Computers" (formerly #317) for more information about this performance factor. Future MPW compiler and library releases will support faster transcendental operations and floating point–to–integer conversions.

Q *In the past we had heard of a problem using calloc and NewPtr in the same program. Is this true?*

A There are a few difficulties, which you can deal with if you need to. The primary problem is that calloc and all the other malloc routines weren't designed for the Macintosh platform. Macintosh memory management is designed around trying to squeeze as much as possible out of a limited memory area, which is why handles are the primary storage scheme in a Macintosh; they can move, and so greatly reduce memory fragmentation. Because the malloc tools return a pointer, they have to be located in a locked block, so they tend to lead to fragmentation if used with any other memory allocation calls (such as NewPtr). For this reason, any use of the malloc suite of allocation calls isn't recommended for Macintosh programs. The only good reason to use them is if you're porting a large body of code from other platforms; in this case, it may be a reasonable tradeoff to keep the old allocation code.

You should also be aware that most of the Macintosh malloc routines never free up memory. When you malloc some space, the routine must first allocate it from the Memory Manager. It allocates a large block of space using NewPtr and divides it internally for distribution in response to malloc calls. If, however, you eventually free all the blocks you allocated from this NewPtr block, the block won't be released to the Memory Manager with the DisposPtr call. This means that once you allocate some memory with malloc, you won't be able to free it and then use that memory from a Macintosh allocation call. Thus, if you had two phases to your program, one of which used the malloc calls extensively and the second which used Toolbox calls, the second phase wouldn't be able to use memory freed by the first phase. That memory is still available in the malloc pool, of course; it simply can't be used by NewPtr or NewHandle. The malloc routines supplied by THINK C work similarly, as described in their *Standard Libraries Reference*. Thus, mixing the C and Macintosh allocation routines requires special care.

Q *Why do I get error -903 (a PPC Toolbox noPortErr) when I send an Apple event to a running application with AESend?*

A The isHighLevelEventAware bit of the sending application's SIZE -1 resource (and SIZE 0 resource, if any) must be set.

Q *Sometimes the Alias Manager mistakes one volume for another. In particular, we're experiencing problems getting aliases to volumes to work correctly with our AppleTalk Filing Protocol (AFP) server volumes. Here's how I can duplicate the problem:*

1. *I mount two server volumes from my AFP server: VolA and VolB.*

2. *I use the Finder to create an alias file for each volume.*
3. *I unmount VolA.*
4. *I open the alias file for VolA. However, when I do this, VolB (which is still mounted) is opened.*

Is this a bug in the Alias Manager or did we implement something the wrong way in our server?

A As noted in the Alias Manager chapter of *Inside Macintosh* Volume VI, the Alias Manager uses three criteria to identify a volume: the volume's name, the volume's creation date, and the volume's type. If the Alias Manager can't find a mounted volume that matches all three criteria, it tries again with just the volume's creation date and the volume's type. This second attempt finds volumes that have been renamed. If that attempt fails, the Alias Manager tries one last time on mounted volumes with the volume's name and the volume's type. If it can't find a mounted volume with those three attempts and the alias is to an AFP volume (a file server), the Alias Manager assumes the volume is unmounted and attempts to mount it.

The problem you're having is probably happening because both volumes have the same creation date and type. That will cause the Alias Manager to mistake VolA for VolB and VolB for VolA when it attempts to match by volume creation date and volume type. You can prevent the Alias Manager from making this mistake by making sure your server volumes all have unique volume creation dates.

This same behavior can be observed when partitioned hard disks use the same volume creation date for all partitions. If one partition isn't mounted, the Alias Manager can mistake one disk partition for another.

Q *I'm looking for a Macintosh Toolbox routine that will allow me to turn down the backlight on a Macintosh PowerBook from within a screen saver to prevent screen burn and save battery life. Is there such a thing?*

A Turning down the backlight won't prevent screen burn. Screen burn can be prevented only by either shutting the system off or letting the PowerBook enter its native sleep mode.

In an RGB monitor the phosphor that illuminates each pixel is what causes screen burn. By setting the pixels to black (the phosphor isn't active) or rapidly changing the colors of an RGB screen (as with a screen saver), you can prevent screen burn. While effective on an RGB display, setting the pixels to black may actually *cause* screen burn on a PowerBook. The reason is that all the

PowerBooks have a liquid crystal display (LCD), which can be burned by white pixels, black pixels, or repeating patterns on the screen over a period of time. For this type of display the only good way to save the screen is to power it off.

Only the Power Manager has access to the chip that shuts the screen off. After a certain amount of time, the Power Manager makes the function calls to put the system to sleep. (These calls are documented in Chapter 31 of *Inside Macintosh* Volume VI.) At this time the Power Manager signals the chip to turn the screen off. There's no direct interface between the user and the chip to achieve this. It's best to let the PowerBook's native screen-saving mechanism (sleep mode, which shuts off the screen) work as is. This also has the benefit of saving the precious battery power that would be used by the screen saver.

By the way, if your PowerBook screen has ghost images because you've left it on too long without going into sleep mode, letting the screen sleep or shutting down your computer for at least 24 hours will probably make the ghost images go away. Although there's no hard and fast rule, usually ghost images caused by your system being on for less than 24 hours won't be permanent if the screen is rested for an equal amount of time. Any ghost images caused by the system being on for greater than 24 hours may be permanent.

Q *How can I call Connect in AppleTalk Remote Access without an existing ARA connection file created by the Remote Access application?*

A This isn't directly possible, because without the ARA connection file your program becomes tied to the underlying link tool. The file was implemented so that in the future, when there are different link tools for the different link types, the program will know the link type and tool, plus associated link-specific data to use. To connect without the ARA connection file requires knowledge of the link tool data structures used by each individual link tool. Because these may change, your code may break.

However, there's a roundabout way of calling Connect. It requires that you first establish a connection using a document created by the ARA application. Next, make the IsRemote call, setting the optionFlags to ctlir_getConnectInfo (see page 11 of the *AppleTalk Remote Access Application Programming Interface Guide)*. This will cause the information necessary to create the remote connection (connectInfoPtr) to be returned. You would then save this connectInfo data in your application, and when you want to connect sometime later, you would pass this data to the Connect call (in the connectInfo field).

Q *When we allocate space for a new file using AllocContig with an argument in multiples of clump size, we should be grabbing whole clumps at a time so that file length (and physical EOF) will be a multiple of clump size. What happens if we truncate a file by* *moving the logical EOF somewhere inside a clump? Inside Macintosh says disk sectors are freed at the allocation block level, so we could have a file whose physical EOF isn't a multiple of clump size, right? Does AllocContig guarantee that the new bytes added are contiguous with the end of the existing file, or only that the newly added bytes are contiguous among themselves? If the logical and physical EOFs aren't the same, does AllocContig subtract the difference before grabbing the new bytes, or do we get the extra bytes (between EOFs) as a bonus?*

A You can create a file whose physical size isn't a multiple of the clump size, if you try. When the file shrinks, the blocks are freed at the allocation level, without regard for the clump size. Therefore, if you set the logical EOF to a smaller value, you can create a file of any physical length.

There's no guarantee that the allocated bytes will be contiguous with the current end of the file. The decisions that file allocation makes are as follows:

- It always attempts to allocate contiguously, regardless of whether you're explicitly doing a contiguous allocation. (If it can't, it fails rather than proceeding if doing an AllocContig.)
- It always attempts to keep the added space contiguous with the existing space, but it will forgo this before it will fragment the current allocation request (regardless of whether you're calling Allocate or AllocContig).

So these are the actions that file allocation will take:

1. Allocate contiguous space immediately after the current physical end of file.
2. Allocate contiguous space separated from the current physical EOF.
3. Fail here if allocating contiguously.
4. Allocate fragmented space, where the first fragment follows the physical EOF.
5. Allocate fragmented space somewhere on the volume.

You don't get "extra" space with AllocContig. It just does a basic allocation but makes sure any added blocks are contiguous. PBAllocContig does *not* guarantee that the space requested will be allocated contiguously. Instead, it first grabs all the room remaining in the current extent, and then guarantees that the remaining space will be contiguous. For example, if you have a 1-byte file with a chunk size of 10K and you try to allocate 20K, 10K-1 bytes will be added to the current file; the remaining 10K+1 bytes are guaranteed to be contiguous.

Q *Inside Macintosh says that ROM drivers opened with OpenDriver shouldn't be closed. However, it seems that any driver opened with OpenDriver should be closed when the application is done. Should our application close the serial port after using it?*

A As a general rule, applications that open the serial driver with OpenDriver should do so only when they're actually going to use it, and they should close it when they're done. (Note that it's important to do a KillIO on all I/O before closing the serial port!) There are a couple of reasons for closing the port when you're finished using it. First, it conserves power on the Macintosh portable models; while the serial port is open the SCC drains the battery. Second, closing the serial port avoids conflicts with other applications that use it. *Inside Macintosh* is incorrect in stating that you shouldn't close the port after issuing an OpenDriver call.

Most network drivers shouldn't be closed when an application quits, on the other hand, since other applications may still be accessing the driver.

Q *We've tried to put old CDs to productive use. We use them for coasters, but you can only drink so many Mountain Dews at once. We've even resorted to using them for skeet-shooting practice. Can you suggest other good uses for my old CDs?*

A It's not well known that stunning special effects in some films, such as *Terminator 2*, were produced with the aid of compact disc technology. For example, the "liquid metal" effect used for the evil terminator was nothing more than 5000 remaindered Madonna CDs, carefully sculpted into the shape of an attacking android. And did you know that dropping a CD into a microwave oven for five seconds or so produces an incredible "lightning storm" effect? (Kids, don't try this at home; we're trained professionals.) For ideas of what *you* can do with old CDs, see the letter on page 5.

Q *I need to launch an application remotely. How do I do this? The Process Manager doesn't seem to be able to launch an application on another machine and the Finder Suite doesn't have a Launch Apple event.*

A What you need to do is use the OpenSelection Finder event. Send an OpenSelection to the Finder that's running on the machine you want to launch the other application on, and the Finder will resolve the OpenSelection into a launch of the application.

As you can see if you glance at the OpenSelection event in the Apple Event Registry, there's one difficulty with using it for remote launching: You have to pass an alias to the application you want to launch. If the machine you want to launch the application on is already mounted as a file server, this isn't important, since you can create an alias to that application right at that moment. Or, if you've connected in the past (using that machine as a server) you can send a previously created alias and it will be resolved properly by the Finder on the remote machine.

However, if you want to launch a file without logging on to the other machine as a server, you'll need to use the NewAliasMinimalFromFullPath routine in the Alias Manager. With this, you'll pass the full pathname of the application on the machine you want to launch on, and the Alias Manager will make an alias to it in the same way it does for unmounted volumes. The obvious drawback here is that you'll need to know the full pathname of the application — but there's a price to pay for everything. The FinderOpenSel sample code on the *Developer CD Series* disc illustrates this use of the NewAliasMinimalFromFullPath routine.

Q *When I try to link my driver in MPW 3.2, it tells me*

```
### Link: Error : Output must go to exactly one segment when using
"-rt" (Error 98)
### Link: Errors prevented normal completion.
```

In all my source files I have #pragma segment Main {C} and SEG 'Main' {Asm} directives. Why is it doing this? What factors determine how segments are assigned (besides the #pragma stuff)? How can I get it to work?

A The problem is probably that you're including modules from the libraries that are marked for another segment. Usually the culprit here is that some of the routines in StdCLib or some other library are marked for the StdIO segment. You can generally fix this by using the -sg option to merge segments, either explicitly by naming all the segments you want to merge, or implicitly by just putting everything into one segment. You probably want to do the latter, because you only want one segment anyway. Thus, what you should do is add the option "-sg Main" to your link line in place of the "-sn Main=segment" option. This will merge all segments into the Main segment, making it possible to link.

Q *How do I count the number of items in a dialog without System 7's CountDITL? My solutions are either messy or dangerous: (1) Fiddle with the dialog's item list, (2) Try to find out which DITL the dialog used and read the count from the DITL resource, or (3) Repeatedly call GetDItem until garbage is returned. :-(*

A It's possible to use the CountDITL function with system software version 6.0.4 or later if the Macintosh Communications Toolbox is installed, because it's included as a part of the Toolbox. It's also possible, as you've found, to use the first two bytes of the DITL resource to get the number of items in the item list (see *Inside Macintosh* Volume I, page 427). If the handle to your DITL resource is defined as ditlHandl, for example, you can get at the number of items as follows:

```
short    **ditlHandl;
ditlHandl = (short **)ditlRez;
itemcount = (**ditlHandl) + 1;
```

Q *How does Simple Player determine whether a movie is set to loop or not? Movie files that are set to loop seem to have a string of 'LOOP' at the end of the 'moov' resource. Does Simple Player check 'LOOP'?*

A Simple Player identifies whether movies are set to loop by looking within the user data atoms for the 'LOOP' atom, as you've noticed. It's a 4-byte Boolean in which a value of 1 means standard looping and a value of 0 means palindrome looping. Your applications should add the user data 'LOOP' atom to the end of the movie when a user chooses to loop. We recommend this method as a standard mechanism for determining the looping status of a movie. If the 'LOOP' atom doesn't exist, there's no looping. The calls you need to access this information are GetMovieUserData, GetUserData, AddUserData, and RemoveUserData, as defined in the Movie Toolbox chapter of the QuickTime documentation. For more information see the Macintosh Technical Note "Movies 'LOOP' Atom."

Q *Calling SetFractEnable seems to force the width tables to be recalculated regardless of the setting of the low-memory global FractEnable. We're calling this routine at a central entry point for any document, as it's a document-by-document attribute. We then unconditionally call SetFractEnable(false) on exit back to the event loop, to be nice to other applications. Calling SetFractEnable(false) seems to trigger the recalculation even though FractEnable is false. What's the best way to get around this?*

A Your observation is correct. The SetFractEnable call stuffs the Boolean parameter (as a single byte) into the low-memory global $BF4 and indiscriminately invalidates the cached width table by setting the 4-byte value at $B4C (LastSpExtra, a Fixed value) to -1. Obviously, it wasn't anticipated that SetFractEnable could be called regularly with a parameter that often doesn't change the previous setting. (By the way, the same observation applies to SetFScaleDisable.)

In your case, you may want to keep track of the fractEnable setting in your application and avoid redundant SetFractEnable calls. (Note that it's not a good idea to use the above insider information and poke at $BF4 and $B4C on your own!)

You don't need to think of other applications when resetting fractEnable; it belongs to those low-memory globals that are swapped in and out during context switches to other applications.

Q *It looks as though the Event Manager routine PostHighLevelEvent could be (ab)used to send low-level messages, like phony mouse clicks and keystrokes. Would this work?*

A No; unfortunately, this won't work. A few reasons why:

- The only applications that will receive high-level events (and their descendants, like Apple events) are applications that have their HLE bit set in their SIZE resource. If you try to send (or post) an HLE to an older application you'll get an error from the PPC Toolbox telling you that there's no port available.
- There's no system-level translator to convert these things. There are currently translators to change *some* Apple events. Specifically, the Finder will translate any "puppet string" event into puppet strings for non-System 7 applications (odoc, pdoc, and quit), but these are *very* special.
- The only way to send user-level events such as mouse clicks through HLEs is to use the Apple events in the MiscStndSuite shown in the Apple Event Registry. And all those events *assume* that the receiving application will do the actual translations to user actions themselves.
- HLEs come in through the event loop. So even if it were possible (through some very nasty patching to WaitNextEvent) to force an HLE into a non–HLE-aware application, the event would come in with an event code of 23 (kHighLevel) and the targeted application would just throw it away.

So the answer is that you can't send user-level events to an HLE-aware application. If you want to drive the interface of an old application in System 7, you have to use the same hacky method you used under all previous systems. This, by the way, is one of the main reasons why MacroMaker wasn't revised for System 7. Apple decided that it wasn't supportable and that we would wait for applications to update to System 7 and take advantage of third-party Apple event scripting systems.

Q *What's the recommended method for allowing an AppleTalk node to send packets to itself using AppleTalk's self-send mode (intranode delivery), assuming customers are running various versions of AppleTalk? There used to be a control panel called SetSelfSend that would turn on AppleTalk self-send mode at startup time. Should we use that control panel or should we use the PSetSelfSend function in our program to set the self-send flag ourselves?*

A AppleTalk self-send mode requires AppleTalk version 48 or greater. You can check the AppleTalk version with Gestalt or SysEnvirons. All Macintosh models except for the Macintosh XL, 128, 512, and Plus have AppleTalk version 48 or greater in ROM.

The SetSelfSend control panel is still available on the *Developer CD Series* disc (Tools & Apps:Intriguing Inits/cdevs/DAs:Pete's hacks-Moof!:SetSelfSend). However, we don't recommend it as a solution if you need to use self-send mode in your program. Instead, you should use the PSetSelfSend function to turn self-send mode on with your program.

AppleTalk's self-send mode presents a problem. Any changes made to the state of self-send will affect all other programs that use AppleTalk. That is, self-send mode is global to the system. Because of this, programs using self-send should follow these guidelines:

- If you need self-send for only a brief period of time (for example, to perform a PLookupName on your own node), you should turn it on with PSetSelfSend (saving the current setting returned in oldSelfFlag), make the call(s) that require self-send, and restore self-send to its previous state.
- If you need self-send for an extended period of time (for example, the life of your application) in which your program will give up time to other programs, you should turn self-send on and leave it on — do not restore it to its previous state! Since other programs running on your system (that aren't well-behaved) may turn off self-send at any time, programs that require self-send should periodically check to make sure it's still on with either PSetSelfSend or PGetAppleTalkInfo. Apple's system software has no compatibility problems with self-send — that is, it doesn't care if it's on or off — so leaving it on won't hurt anything.

Q *In a version 2 picture, the picFrame is the rectangular bounding box of the picture, at 72 dpi. I would like to determine the bounding rectangle at the stored resolution or the resolution itself. Is there a way to do this without reading the raw data of the PICT resource itself?*

A With regular version 2 PICTs (or any pictures), figuring out the real resolution of the PICT is pretty tough. Applications use different techniques to save the information. But if you make a picture with OpenCPicture, the resolution information is stored in the headerOp data, and you can get at this by searching for the headerOp opcode in the picture data (it's always the second opcode in the picture data, but you still have to search for it in case there are any zero opcodes before it). Or you can use the Picture Utilities Package to extract this information.

With older picture formats, the resolution and original bounds information is sometimes not as obvious or easily derived. In fact, in some applications, the PICT's resolution and original bounds aren't stored in the header, but rather in the pixel map structure(s) contained within the PICT.

To examine these pixMaps, you'll first need to install your own bitsProc, and then manually check the bounds, hRes, and vRes fields of any pixMap being passed. In most cases the hRes and vRes fields will be set to the Fixed value 0x00480000 (72 dpi); however, some applications will set these fields to the PICT's actual resolution, as shown in the code below.

```
Rect        gPictBounds;
Fixed       gPictHRes, gPictVRes;

pascal void ColorBitsProc (srcBits, srcRect, dstRect, mode,
   maskRgn)
BitMap      *srcBits;
Rect        *srcRect, *dstRect;
short       mode;
RgnHandle   maskRgn;
{
   PixMapPtr   pm;
   pm = (PixMapPtr)srcBits;
   gPictBounds = (*pm).bounds;
   gPictHRes = (*pm).hRes;    /* Fixed value */
   gPictVRes = (*pm).vRes;    /* Fixed value */
}
void FindPictInfo(picture)
PicHandle   picture;
{
   CQDProcs    bottlenecks;
   SetStdCProcs (&bottlenecks);
   bottlenecks.bitsProc = (Ptr)ColorBitsProc;
   (*(qd.thePort)).grafProcs = (QDProcs *)&bottlenecks;
   DrawPicture (picture, &((**picture).picFrame));
   (*(qd.thePort)).grafProcs = 0L;
}
```

Q *The code I added to my application's MDEF to plot a small icon in color works except when I hold the cursor over an item with color. The color of the small icon is wrong because it's just doing an InvertRect. When I drag over the Apple menu, the menu inverts behind the icon but the icon is untouched. Is this done by brute force, redrawing the small icon after every InvertRect?*

A The Macintosh system draws color icons, such as the Apple icon in the menu bar, every time the title has to be inverted. First InvertRect is called to invert the menu title, and then PlotIconID is called to draw the icon in its place. The advantage of using PlotIconID is that you don't have to worry about the depth and size of the icon being used. The system picks the best match from the family whose ID is being passed, taking into consideration the target rectangle and the depth of the device(s) that will contain the icon's image.

The Icon Utilities call PlotIconID is documented in the Macintosh Technical Note "Drawing Icons the System 7 Way" (formerly #306); see this Note for details on using the Icon Utilities calls.

Q *The cursor flashes when the user types in TextEdit fields in my Macintosh application. This is done in TEKey. I notice that most programs hide the cursor once a key is pressed. I don't care for this because then I have to move the mouse to see where I am. Is this a typical fix for this problem and an accepted practice?*

A There's very little you can do to avoid this. The problem is that every time you draw anything to the screen, if the cursor's position intersects the rectangle of the drawing being done, QuickDraw hides the cursor while it does the drawing, and then shows it again to keep it from affecting the image being drawn beneath it. Every time you enter a character in TextEdit, the nearby characters are redrawn. Usually this is invisible because the characters just line up on top of their old images, but if the cursor is nearby and visible, it will flicker while it's hidden to draw the text. This is why virtually all programs call ObscureCursor when the user types. Also, most users don't want the image of the cursor obscuring text they might be referring to, yet they don't want to have to move it away and then move it back to make selections. Because it's so commonplace, hiding the cursor probably won't bother your users; in fact, they might very well prefer the cursor hidden. This, combined with the fact that there's very little you can do to help the flickering, suggests that you should obscure the cursor while the user types.

Q *We're using Apple events with the PPC Toolbox. We call StartSecureSession after PPCBrowser to authenticate the user's identity. The user identity dialog box is displayed and everything looks good. However, in the first AESend call we make, the user identity dialog is displayed again. (It isn't displayed after that.) Why is this dialog being displayed from AESend when I've already authenticated the user identity with StartSecureSession?*

A First, a few PPC facts:

- When a PPC session is started, StartSecureSession lets the user authenticate the session (if the session is with a program on another Macintosh) and returns a user reference number for that connection in the userRefNum field of the PPCStartPBRec. That user reference number can be used to start another connection (using PPCStart instead of StartSecureSession) with the same remote Macintosh, bypassing the authentication dialogs.

- User reference numbers are valid until either they're deleted with the DeleteUserIdentity function or one of the Macintosh systems is restarted.
- If the name and password combination used to start a session is the same as that of the owner of the Macintosh being used, the user reference number returned refers to the default user. The default user reference number normally is never deleted and is valid for connections to the other Macintosh until it's deleted with DeleteUserIdentity or one of the Macintosh systems is restarted.

With that out of the way, here's how user reference numbers are used when sending high-level events and Apple events: When you first send a high-level event or an Apple event to another Macintosh, the code that starts the session with the other system doesn't attempt to use the default user reference number or any other user reference number to start the session, and it doesn't keep the user reference number returned to it by StartSecureSession. The session is kept open for the life of the application, or until the other side of the session or a network failure breaks the connection.

When you started your PPC session, StartSecureSession created a user reference number that could be used to start another PPC session without authentication. However, the Event Manager knows nothing of that user reference number, so when you send your first Apple event, the Event Manager calls StartSecureSession again to authenticate the new session. Since there isn't any way for you to pass the user reference number from the PPC session to the Event Manager to start its session, there's nothing you can do about this behavior.

Q *How can I make my ImageWriter go faster?*

A To make your ImageWriter go blazingly fast, securely tie one end of a 12-foot nylon cord around the printer and the other end around your car's rear axle. If your car has a manual transmission, hold the clutch in and race your car's engine until the tachometer is well into the red zone. Slip the clutch and off you go! If your car has an automatic transmission, you can approach the same results by leaving plenty of slack in the rope before peeling out.

Have more questions? Need more answers? Take a look at the Q & A Technical Notes on the *Developer CD Series* disc and the Dev Tech Answers library on AppleLink.•

KON & BAL'S PUZZLE PAGE

A MICRO BUG

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. Try to guess this one before BAL does. To figure out your score, see "Scoring" at the end.

KONSTANTIN OTHMER AND BRUCE LEAK

KON Have you heard of Spaceward Ho!?

BAL Yeah, it's that awesome conquer-the-galaxy game from Delta Tao. That game has done more to hurt productivity around here than pinball.

KON After they released it, they got several calls complaining about a crash. They tried to reproduce the crash but couldn't.

BAL They don't have that SADE MultiFinder installed, do they?

KON Very funny.

BAL How is their configuration different from the configuration of customers with the problem?

KON Everyone who complained had a 4-meg IIsi, ci, or fx. And the Delta Tao folks tested those configurations.

BAL Hmmm. How does it crash? Can you get into MacsBug?

KON That's part of the problem, the customers who have the crash aren't programmers and don't have MacsBug. The crash is with an Error 01, a bus error.

BAL Well, find one of the machines it crashes on, install MacsBug, and see what's wrong. How hard can it be?

KON So you fly to Bismarck, North Dakota, and install MacsBug, and it doesn't crash anymore. Pretty hard, I guess.

BAL Hmmm. Just MacsBug? Are there any INITs running?

KON The machine has only MacsBug, nothing else.

KONSTANTIN OTHMER AND BRUCE LEAK are basically slackers who go on way too many vacations. Unfortunately, they write buggy code and there are always a number of bugs that they need to fix on their return. But in true slacker style, they wouldn't think of fixing their own bugs. Enter the Puzzle Page, a sly coverup for getting someone else to solve these problems. Instead of fighting through buggy code with MacsBug, they call each other looking for easy answers. To keep pace with their bugs, they're lobbying the *develop* staff to do a whole issue of just Puzzle Pages.•

BAL And you never set a breakpoint, or an A-trap break, or anything?

KON Nope.

BAL Do you have a FirstTime macro?

KON Nope.

BAL So how could MacsBug be interfering?

KON I can’t help you there. It’s your puzzle.

BAL Well, MacsBug initializes some low-memory values and rearranges things above BufPtr. Is the app doing anything funny that might depend on some low mems?

KON The app follows every programming convention dictated by *Inside Macintosh* and the Developer Support Center. They even follow every human interface guideline and . . .

BAL Yeah, yeah, yeah. Impossible. So MacsBug is installed, but it’s never invoked.

KON Yep.

BAL What’s the app doing when it crashes?

KON It’s in the middle of a bunch of calculations — you know, how many ships got destroyed in battle, how fast planets’ populations are growing, what the computer players are doing, that kind of thing.

BAL Well, MacsBug causes the app to launch in a different place.

KON OK.

BAL MacsBug loads above BufPtr, so everything else loads lower. Maybe the app reads past the end of its heap. When MacsBug is in, it’s lower in the heap, so the app reads somewhere in MacsBug territory. When MacsBug is out, the app reads past the end of RAM and causes a bus error.

KON Nice theory. But how do you verify that that’s the problem without MacsBug?

BAL Launch another app first.

KON Then the Ho! will load even lower in memory. It won’t crash.

BAL Use MicroBug.

KON You mean that thing that comes up when you push the NMI switch and MacsBug isn’t installed? Where is that documented?

BAL I don’t know. It can’t be too hard to figure it out, though.

KON Well, the only command I know is G for “Go.” What else will it let me do?

BAL You can look at memory and registers, you can set the PC, and you can even exit to the shell. Let's try a Total Display, TD. MicroBug responds with this:

```
000C30  0000 0000 0074 0000  FFFF 0100 0000 00C4
000C40  0000 FFFF 0000 0000  00AD E5D7 0074 0000
000C50  006E B2D0 0074 0A80  006E 9EB8 0057 0308
000C60  0000 0000 0074 0BAC  006E 49F8 006E 49E0
000C70  000A D96A 2014 0000  0000 0000 0000 0000
000C80  0000 0000 5444 0020  0020 0020 0020 0020
```

KON It looks like it's dumping memory from C30.

BAL Yeah, from SysEqu.a we see that C30 is SEVarBase. The system exception vars go up to CBF. I guess that's where the exception vectors dump the processor state when an exception occurs.

KON Since the system sets up the SEVars, they're set up on any exception regardless of the debugging environment. Using MacsBug, we can figure out that the first two lines are registers D0-D7, the next two lines are A0-A7, then the PC, then the status register, then what?

BAL I don't know, but at C84, it looks like what we typed: TD.

KON You could read a book written in ASCII!

BAL Let's try something else, maybe it can do math. Let's try DM PC-10.

KON It works.

BAL Yeah. In addition to the PC, it knows registers as RA0 or RD0 (but you set registers with a line like D0 = 5, not RD0 = 5). You can set memory using SM.

KON Anyway, back to the Ho!

BAL So in the Ho! I can look at the PC and the registers and figure out that it's looking past the end of memory.

KON You can't do an IL or an IP, so you can't prove that bogus values in a register are causing the bus error.

BAL I go into MacsBug on my PowerBook and disassemble the code with the DH command.

KON How do you find the problem code in the source?

BAL I pattern-match using the Find command on the PowerBook. Once I find the problem in MacsBug on the PowerBook, I'm golden.

KON Right! Here's the scoop: One of their pointers got messed up and they were reading off the end of their heap. The value they read had only a minor impact on the calculations, so no one noticed the problem. When MacsBug

SCORING

If you stick with MacsBug and never even try MicroBug, score 25.
If you figured out the bug before BAL did, score 50.
If you start to use MicroBug and like it better than MacsBug, score 75.
If you start to play Spaceward Ho! regularly, and like it better than MicroBug, score 100.•

was in, they were reading in MacsBug's code space, which is a valid address and didn't cause a bus error. The reason it was reported on 4-meg IIsi's, ci's, and fx's is that only '030 or '040 machines that have the ci-class ROM cause bus errors when reading a valid RAM address that doesn't have RAM installed.

BAL And reading off the end of RAM on an 8-meg machine in 24-bit addressing mode just reads the ROM, which is valid.

KON Instead of this MicroBug detour, you could just write a flag value on the screen from various interesting places in the source. The flag value when you crash tells you where you were last.

BAL Yeah, but that's been done before. And it doesn't give us a good excuse to discuss MicroBug.

KON OK, Mr. MicroBug, what's the fewest keystrokes you can use to do an ExitToShell from MicroBug?

BAL Well, ExitToShell is Toolbox trap A9F4. The Toolbox trap table begins at $E00, so you can calculate the address of the trap and then use the G command.

KON Once you have the address, that's a minimum of seven keystrokes. You like to type a lot.

BAL I need some time to think about that one.

KON While you're thinking, how do you restart from MicroBug?

BAL Let's just leave everyone in suspense until next time.

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 the *Developer CD Series* disc.•