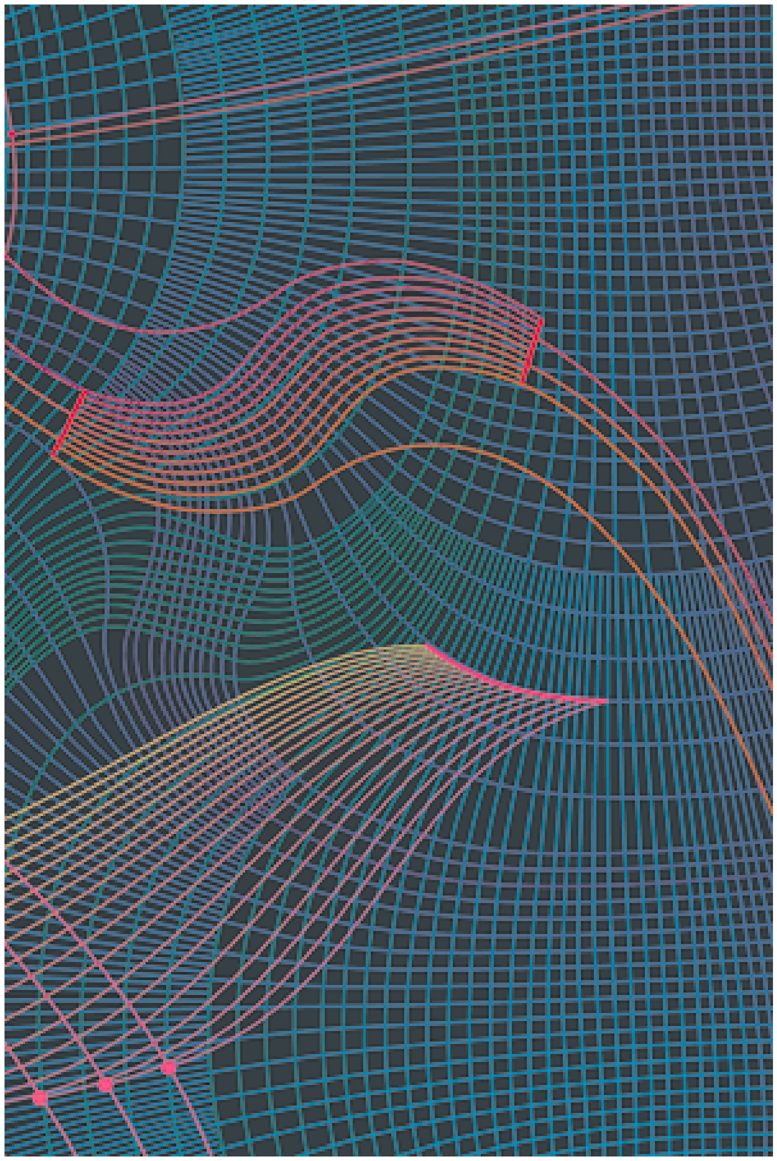
develop

The Apple Technical Journal



Issue 6 Spring 1991

Apple Computer, Inc.

EDITORIAL

Editor-in-Cheek *Caroline Rose*

Spirited Guide *Louella Pizzuti*

Technical Buckstopper *Dave Johnson*

Managing Editor *Monica Meffert*

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

Editorial Assistant *Helen Stea*

Indexer *Ira Kleinberg*

Manager, Developer Technical Communications *David Krathwohl*

ART & PRODUCTION

Production Manager *Hartley Lesser*

Art Director *Diane Wilcox*

Design *Joss Parsey*

Technical Illustration *Don Donoughe and Diane Wilcox*

Formatting *Automatrix*

Printing *Craftsman Press*

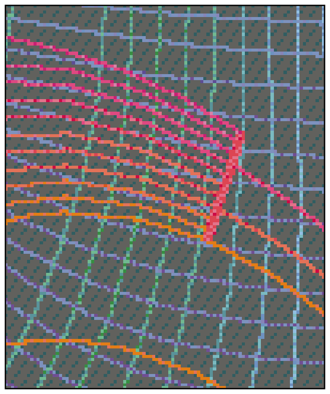
Film Preparation *Aptos Post Inc.*

Production *PrePress Assembly*

Photographer *Ralph Portillo*

Circulation Management *David Wilson*

Online Production *Cassi Carpenter*



Cleo Huggins created this cover with Adobe Illustrator®. A theoretical Klein plane was chosen to represent the complex topology on which threads of execution operate.

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

CONTENTS

LOUELLA PIZZUTI PASSES TO CAROLINE ROSE

Dear Readers,

Starting with this issue of *develop*, Louella has passed the editorial baton to yours truly, Caroline Rose. If the name sounds familiar, it's because you may have seen it in the credits for *Inside Macintosh*, Volumes I-III. At one point during my three-and-a-half-year stint as the primary author and editor of that magnum opus, I interviewed an inexperienced but delightfully cocky young woman who convinced me that she could do as good a job as anyone of formatting the *IM* files on the Macintosh® (they were created on the Apple® III). She was hired, and ended up being a great help to the project as well as a great friend. You'll also find her name in the credits: "assisted by" none other than Louella Pizzuti.

But the time came for me to move on. I went looking for jobs and found the next company I would work for (puns intended). After four years as that next company's Manager of Publications and a year as its Editor in Chief, I was ready to move on again. I put the word out at Apple and got a phone call from Louella, with whom I had fallen woefully out of touch over the years. Now the friendship has been renewed and we're working together once again—this time with her as my boss!

After a brief orientation (and a wonderful "welcome" lunch) on my first day back at Apple, I was steered into a meeting at which I had to immediately make my first decision as *develop*'s new Editor in Chief. Actually, it was sort of a trick question, since everyone present had already agreed on the answer. Fortunately, I was of like mind, and as a result we're calling this simply Issue 6 rather than continue with the previous issue's use of "Volume 2" (which snuck in while Louella was out with a bad back). So for those of you who are collecting the whole set (knowing it will be of inestimable value someday), take note that the previous issue's number should be reassigned according to the following formula:

```
int
CalculateIssueNumber(int Volume, int Issue)
{
      return ((Volume – 1) * 4) + Issue;
}
```

CAROLINE ROSE has been writing computer documentation ever since "a computer at your fingertips" meant timesharing a mainframe. After a seven-year digression into programming, she returned to her first love—writing—and was given the opportunity to document the inner workings of a funny little computer named Macintosh. The result was a three-volume tome that, in its hard-cover edition, was the first technical manual to qualify as a lethal weapon (and we don't mean by boring people to death). In what proved to be another digression, she left Apple to launch NeXT's documentation effort. Returning after a five-year hiatus, she's thrilled to be back at Apple among friends and foe alike. For pleasure outside of work, Caroline reads voraciously, swims fanatically, dances, sings, plays Scrabble, hugs her cat, and much more.

Or, for you Pascal types:

```
FUNCTION CalculateIssueNumber (Volume, Issue: INTEGER): INTEGER;
BEGIN
      CalculateIssueNumber := ((Volume - 1) * 4) + Issue;
END;
```

You may notice other changes introduced in this issue. We hope you'll agree they're changes for the better; if not, be sure to let us know. Please also let us know if you'd be interested in writing an article for *develop* yourself. Do you have any nifty code you'd like to share with the rest of the Apple developer world? Have you yearned to have your own silly bio grace these pages (not to mention your photo)? Think of how pleased your mother would be to receive a copy.

I'll leave you with a riddle: I entered this entire editorial without pressing a single key on the keyboard or clicking the mouse button. I was as quiet as a mouse (the furry kind). How did I do this? And furthermore, why? Stay tuned for the answer in our next issue.

CRose

Caroline Rose
Editor

SUBSCRIPTION INFORMATION
Use the order form on the last page of this issue to subscribe to *develop.* Please address all subscription-related inquiries to *develop,* Apple Computer, Inc., P.O. Box 531, Mt. Morris, IL 61054 (or AppleLink DEV.SUBS).•

BACK ISSUES
Back issues of *develop* are available through APDA® (see inside back cover for APDA information) and are also on the *Developer Essentials* CD-ROM disc.•

LETTERS

•I encountered a technical error in the Macintosh Q & A section in the latest issue of *develop*. The question was: "Is the maximum size for global and local data still 32K?" The answer stated that the 32K limit for local (stack) data "is basically due to the Motorola processor architecture." As stated in the answer, the LINK instruction is limited to a 16-bit offset. Unstated in the answer is that the compiler can easily work around this limitation. Suppose your program declares 50K bytes of local data. The compiler should generate a LINK instruction for the first 32K bytes and then adjust the stack for the remaining 18K bytes needed by the routine. If the program is compiled specifically for the 68020 or 68030, the compiler can issue a LINKL (link long) instruction, which allows a full 32-bit offset.

—Chuck Lins

Thanks for the clarification. You're absolutely right, but of course for most of us it doesn't make any difference; we don't write our compilers, we're slaves to them. I guess the answer should have read "is basically due to the way that current compilers handle local data." All you aspiring compiler writers, take note.

—Dave Johnson

•After receiving *Gorillas in the Disc (Developer CD Series* Volume VI) a few months ago, I checked out the electronic versions of the *develop* issues that we did not have in hardcopy, and came across something that prompted me to write. I was reading the letters in *develop* Issue 2 (April 1990) and saw a letter and response in which you mentioned the "ever-popular audio track" from the *develop* CD. Because we are an Apple Partner, our copy of *develop* contains a card telling us that Developer Essentials will be included in a folder on our *Developer CD Series* disc. Imagine my disappointment when I realized that the disc Apple Partners received contained no audio track. What a crime to deprive all us Partners of such a bonus! Please tell your people that Partners want their audio!

—Bill Stoker

The Developer CD Series disc contains a superset of what's on the disc that's bound into develop *(now called the Developer Essentials disc). Volume III of the Developer CD Series ("A Disc Called Wanda") corresponds to* develop *Issue 2 and does indeed contain the ever-popular audio track, as part of the CD Audio Toolkit demo. Were you looking for an audio track on the latest Developer CD Series disc? If so, you would in fact not find one there; there wasn't one on the corresponding Developer Essentials disc, either. Rest assured that we won't deprive Apple Partners of any of the goodies we provide on the Developer Essentials disc!*

—Caroline Rose

•First let me say that *develop* is great!

I read on page 5 of the latest issue that your group is now responsible for the *Developer CD Series*. *Gorillas in the Disc* is a bit of a disappointment. Not a lot of really new stuff on it. That's not your fault, I know. There must be neat stuff floating around in Apple somewhere!

Someone decided that the Q & A Stack should contain both Macintosh and Apple IIGS stuff. Boo, hiss! I tend to read through that stack looking for stuff I

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

don't know, before I know I need to know it! I was really confused by things that I had never heard of on the Macintosh before, only to find they were on the Apple IIGS! Arrggh! Can you please separate the IIGS stuff and the Macintosh stuff into two stacks? And can you please go back to the format of putting dots beside the new stuff?

The X Ref stuff is kinda neat. Can't wait for 7.0 aliasing!

—Scott Anguish

Thanks a lot for your letter and your words of encouragement. My group contributes to the Developer CD Series, but we're not actually responsible for it. Besides adding more new "neat" stuff, is there anything we could do to improve the disc? We're always looking for suggestions!

Sorry to hear that adding Apple IIGS stuff to the Q & A stack confused you so much (but I'm awfully glad to hear that you're browsing the Q & A stack!). We put the IIGS info in the stack hoping that Macintosh developers would see how similar the two toolboxes are and perhaps get some extra mileage from their work by porting their Macintosh application to the IIGS. It sounds like it's more confusing than helpful, so we'll reevaluate our decision. What do the rest of you think about combining IIGS and Macintosh information in one stack?

In reply to your request for dots: The latest Q & A Stack has been reworked so much it's practically all new, but in the future you can expect to once again see dots beside new material. Also note that, starting with the latest stack, each card will show the date of the last modification.

—Louella Pizzuti

•Just a quick note to congratulate you and your staff on another fine issue of *develop* (and in the hopes that I won't see a survey). I read it cover to cover, and thoroughly enjoyed it, especially the Macintosh Q & A. I kept turning the page expecting to have seen the last question answered, but lo and behold, there were more questions. It was better than Jeopardy. You really outdid yourselves! Keep up the wonderful work, and please consider starting a new column: a soap opera about the dogcow.

By the way, whose nose is lighting up the cover?

—Robert H. Zakon

Thanks for the letter. If we ever do a survey (and my boss has been pushing me to do one ever since Issue 2 came out), I'll make sure that the surveyors know you're officially exempt.

The nose on the cover belongs to Cleo Huggins, who has done all our covers and nose what she's doing.

—Louella Pizzuti

THREADS ON THE MACINTOSH

Threads are a great way to improve the performance and simplify the design of programs. Apple's Advanced Technology Group developed a Threads Package to implement this programming technique on the Macintosh. This article explains how you can use this package to incorporate threads in your own code.

MICHAEL GOUGH

The idea for the Threads Package arose during the design phase of some scientific visualization software, when we discovered that some of the applications we were working on needed a way to juggle several simultaneous activities. It quickly became clear that the Macintosh run-time environment posed some serious obstacles to anyone wanting to implement threads on the Macintosh. With some effort, we were able to come up with workarounds that made the use of threads with the Macintosh OS relatively painless.

These workarounds are the main subject of this article. After briefly introducing the purpose and mechanics of threads in general, the article presents some specific details of the Macintosh threads implementation as it currently stands. A summary of the functions in the Threads Package appears at the end of the article. The Threads Package itself and several simple example programs can be found on the *Developer Essentials* disc for this issue.

The Threads Package was developed as a means to an end, and it's by no means the last word on threads for the Macintosh. We welcome any suggestions you may have for improvements.

WHAT THREADS DO

Suppose you want to write an AppleLink®-like communications program. You'd like to write the program so that while it's downloading a file, it can also print an existing file and allow the user to write a new message. A typical program can perform only one of these functions at a time, displaying the watch cursor until the task is completed. What's needed is some technique for allowing the program to perform these tasks concurrently.

MICHAEL GOUGH is a designer in Apple's User Programming Group. We'd tell you what he's up to these days, but it's so secret we'd have to kill you if we did. Before coming to Apple, Michael worked at STX as a NASA contractor, designing scientific data visualization systems. He is best known as the designer and implementor of CDF, a "mini-database" that NASA uses to store data from dozens of spacecraft. Michael developed software used by NOAA's fleet of oceanographic vessels to map the ocean floor. He also worked as a contractor to the United Nations World Meteorological Organization, so if you have any problems with the weather, now you know who to blame. While he was there, he developed real-time satellite tracking and data ingest systems for the TIROS-N, GOES, and GMS spacecraft, and conducted training and

WHY THREADS ARE IMPORTANT IN THE SYSTEM 7.0 ERA

Interprocess communication (IPC) is one of the most compelling reasons why threads are going to become increasingly important in the future. This became clear to a group of us working in Apple's Advanced Technology Group when we observed that a client and a server application communicating via IPC could easily get into a deadly embrace. A client would ask the server application a question and would wait for an answer before continuing. Unfortunately, sometimes the client would wait forever for the answer. What happened was that the server needed to ask its own question of the client before answering the client. However, the client was monitoring exclusively for a response to its question and would ignore the server's question. The client needed to answer the server's incoming question before it could get an answer to its own question. Both client and server would be stopped dead waiting for the other to respond.

In a sense, the Threads Package exists because the problem described here was intractable without threads. The application must be both a client and a server. It must be able to simultaneously handle incoming questions and wait for incoming answers. Other approaches to doing this, such as idle procs, skirted the core of the problem and led to code complexity that was unmanageable. Idle procs push too much of the problem onto the application programmer, who already has enough to worry about.

The threads solution is even more important now that IPC has been integrated into the Macintosh OS in System 7.0. As more programmers will have access to IPC because of System 7.0, they will need this elegant method of achieving concurrency.

Programmers have often tried to achieve concurrency through the use of idle procs. For your communications program, for instance, you could write the downloading, printing, and text entry tasks as idle procs. While the download procedure is executing, it could regularly call a printing idle proc to send a few lines of a message to the printer. The download procedure could also periodically call an editing procedure to allow the user to enter text for a new message in a window.

But think of the tremendous effort involved in writing the program so that it can switch among these tasks. Every task would have to save variables each time it returns so that it could resume where it left off. Most complex functions would not be able to contain deep levels of nesting because that would make it impossible to freely return to the caller at any time. In fact, you'd have to divide most functions into inconveniently small chunks so that you could juggle between them. The net result is that the modularity of your program would be destroyed, and you'd have a tremendous programming headache on your hands.

Threads are a much better technique for achieving concurrency than idle procs. When your program uses threads, it's like a mind that can have several trains of thought simultaneously. A program using idle procs, in contrast, is like a mind with a single train of thought that must constantly interrupt itself to attend to side issues.

Note that there's a difference between multithreaded programs and multitasking systems. Multitasking is the ability to run more than one application at once, but each

installation in Beijing and Buenos Aires. In Beijing he used his knowledge of electronics, computer science, math, and Scotch tape to successfully complete the installation—just goes to show that you never quite know what the right tools for the job are going to be. (Here at Apple, we make sure he always has plenty of office supplies—just in case.) •

application can still only do one thing at a time. In other words, concurrency is happening at the system level. A multithreaded application performs concurrent tasks within the same program; concurrency happens at the program level. Of course, it's possible to have a multitasking environment in which threaded programs run.

IDLE PROCS VERSUS THREADS

Idle procs have traditionally been used to approach thread-like functionality. This involves writing a piece of code to handle a particular task and installing it in a queue of things that get called periodically. Thereafter, the flow of control pulses through the routine, which can do some finite amount of work and then return, so that other idle procs can get pulsed.

This approach results in several gnarly coding problems. The most serious is that the pulsed routine, which is attempting to execute some algorithm, must return to its caller at inopportune moments. Imagine that you're marching through a deeply nested piece of code and you want to relinquish control when you reach a certain point. With the pulsing approach, you must return to the caller from deep within the nested code. You could put in a return statement, but the problem would be that when it's time to pick up where you left off, you would have to magically jump back into the code after the return statement on the next pulse. Obviously, this is not a simple thing to do when you have to bypass several layers of conditionals and loops.

The magic of the Threads Package is that it allows you to avoid these problems: you can leave a complex function and resume execution of it precisely where you left off. With idle procs, on the other hand, you're forced to completely redesign the algorithm. You must give the algorithm an "inside out" appearance: code that was in the most deeply nested part of the algorithm now appears near either the top or the bottom of the routine. You may even have to break your routine into several smaller functions that are run in sequence. But doing these things will negate the natural top-down structure of a routine. It's a mess.

HOW THREADS WORK

When writing multithreaded code, you must let go of old ideas about how the machine executes your program. Instead of a single program counter marching through your code, in a sense you now have many. While the idea of multiple program counters may sound complex, you don't have to relearn programming. You just need to be aware that the main train of execution in a program is itself a thread and that all threads must relinquish control to each other. You also have to remember to share globals and heap objects that you used to access exclusively.

Here's a sample program that shows how simple it is to use threads. The program is a modified version of the ever-popular SillyBalls. Unmodified, the program opens a window and draws colored balls into it until the main event loop detects that the mouse button is down. This new version forks a thread that beeps while the balls are being drawn.

```
main()
{
   ThreadHandle beepThread;

   Initialize();

/* The InitThreads call initializes the Threads Package, converting
the original thread of execution into a swappable thread. */

   InitThreads(nil, false);

/* This code forks a thread that beeps 30 times, and then quits. */

   if (InNewThread(&beepThread, kDefaultStackSize))
      {
      long i;
      for (i=0; i<30; i++)
         {
         SysBeep(120);
         Yield();
         }
      EndThread(beepThread);
      }

/* Here's the main event loop. The only change is the new call to
Yield. */

   do
      {
      Yield();
      NewBall();
      } while (!Button());

/* This call to ExitThreads waits for all threads to die before
allowing the program to terminate. */

   ExitThreads();
}
```

The InitThreads call is made at the beginning of the program. It initializes threads and converts the original thread into something that can be swapped by the Threads Package. Once this call is made, you can fork other threads.

In this example, execution from the original thread enters the InNewThread procedure. Two threads leave the procedure, but at different times. The original thread goes in and is cloned before coming out. A new thread now exists, but it hasn't started execution yet. InNewThread tests whether the current thread's ID is that of the new thread, beepThread, and returns a Boolean indicating the result of this test. It's essentially supplying an answer to the question "Am I running the new thread?" Since the original thread is still the current thread, it returns from InNewThread with a value of false, thus skipping over the code contained in the IF block. It continues execution by entering the main event loop, drawing balls, and calling the Yield function. Each time it calls Yield, it politely gives control to other threads that may want time to execute.

On the first call to Yield, the newly cloned thread returns from the call to InNewThread with a value of true, indicating that this is the new thread and not the original. The new thread enters the block of code associated with the IF statement and begins executing the loop, which beeps and yields 30 times. Each call to Yield exchanges control with the main event loop. The new thread lives out its life within the confines of the IF block. After completing its task, it calls EndThread and dies.

The conditions for terminating these two threads are different: the beeping thread ends after 30 iterations; the original thread ends when the user presses the mouse button. The call to ExitThreads at the end of the program ensures that all threads have completed before the program terminates.

SEMAPHORES

With multiple threads running around in your program, it's possible for them to get in each other's way. The Threads Package provides a semaphore mechanism to help you manage this problem. The problem occurs when two threads compete for a resource. Two threads that are executing at the same time may each want exclusive use of the same device, file, or memory location.

To deal with this situation, you assign a semaphore to control access to this resource. Then, when you write the thread that uses the resource, you always make sure that the thread "grabs" the semaphore. After you're done with the resource, you "release" the semaphore.

What happens if a thread tries to grab a semaphore that has already been grabbed? The thread goes to sleep, waiting in a queue associated with the semaphore. When the semaphore does become available, the sleeping thread wakes up with control of the semaphore, completely unaware that it had to wait in the queue. It continues executing code as usual, and releases the semaphore when it's done, thus giving other threads an opportunity to use the resource.

Below is a small example program that demonstrates the behavior of semaphores. It's very similar to the first example, except that the beeping thread grabs a semaphore before beeping 4 times and then releases it. A call to Yield was inserted within this inner loop just to demonstrate that even though there is a call to Yield in the loop, no balls are drawn during this time. This is because the code that draws the balls grabs the semaphore too. When it gets control of the semaphore, it draws 20 balls before letting go. After you release a semaphore, you still have to call Yield before other threads will get control.

```
main()
{
   ThreadHandle     beepThread;
   SemaphoreHandle  aSemaphore;

   Initialize();

/* The InitThreads call initializes the Threads Package, converting
the original thread of execution into a swappable thread. */

   InitThreads(nil, false);
   aSemaphore = NewSemaphore();

/* Fork the beeping thread. */

   if (InNewThread(&beepThread, kDefaultStackSize))
      {
      long i,j;
      Yield();
      for (i=0; i<10; i++)
         {

/* Grab the semaphore, beep 4 times, and release the semaphore. */

         GrabSemaphore(aSemaphore);
         for (j=0; j<4; j++)
            {
            SysBeep(120);
            Yield();
            }
         ReleaseSemaphore(aSemaphore);
         }
      EndThread(beepThread);
      }
/* Here's the main event loop. */

   do
      {
      long j;
      Yield();

/* Grab the semaphore, draw 20 balls, and release the semaphore. */

      GrabSemaphore(aSemaphore);
      for (j=0; j<20; j++)
         {
         NewBall();
         Yield();
         }
      ReleaseSemaphore(aSemaphore);

      } while (!Button());

/* This call to ExitThreads waits for all threads to die before
allowing the program to terminate. */

   ExitThreads();
}
```

IMPLEMENTING THREADS ON THE MACINTOSH

After examining the ramifications of implementing threads in the Macintosh run-time environment, we identified three serious problems:

- non-reentrant Toolbox and application code
- Toolbox use of memory between the stack and the heap
- segment unloading

Although the Threads Package minimizes the impact of these problems, you must still deal with some special coding issues when writing programs that use threads.

NON-REENTRANT TOOLBOX AND APPLICATION CODE

When you develop code that uses threads, it's important to write reentrant code. This is a fancy way of saying that your threads must not interfere with each other. A common way in which threads do interfere with each other is in the use—or misuse—of global variables.

The basic problem can be described as follows: Your thread is merrily running along, and it politely yields control to the other threads. When it gets control again, the other threads may have unexpectedly changed some global variables, causing your thread to crash and burn, or behave in an unexpected manner.

Let's illustrate this problem with a realistic example. Suppose you want two windows in your application, and you want to have some drawing going on in each of them simultaneously. Naturally, you would start two threads that draw in the two respective windows. Unfortunately, when you run the program, you find that both of the threads end up drawing in the same window.

What happened? The first thread sets its grafPort to the grafPort of the first window. When the first thread yields control to the second thread, the second thread changes the grafPort to point to *its* window. Finally, when the first thread gets control again, the grafPort is still pointing to the second window.

You might attempt to solve this problem by placing code that saves and restores your grafPort before and after your call to Yield. This approach may appear to work, but watch out! There may be other calls to the Yield function in routines that your thread is calling. You would have to make sure your save-and-restore code surrounds every one of these calls as well. This would be cumbersome, to say the least.

A safer solution to the reentrancy problem is simply to write reentrant code from the beginning. In other words, just don't misuse global variables. But alas, millions of lines of code have already been written for the Macintosh with globals galore. The Macintosh Toolbox itself is on the whole non-reentrant. For instance, in the above example, the grafPort global is referenced not just in the application but in the Toolbox itself. It would be unrealistic to expect reentrancy problems in Toolbox and application code to vanish overnight.

To get around all this, the Threads Package provides an innovation called *customizable swapping behavior*. To understand how this behavior works, you must first know a little bit about the thread structure.

The thread structure contains additional fields for the custom procedures that the Threads Package uses to control a thread. Figure 1 illustrates these fields.

You implement the customizable swapping behavior by writing custom routines that carefully set up a thread's globals when the thread swaps in and save these values before the thread swaps out. You assign these routines to the fields in the thread structure, so that the Threads Package can automatically call these routines for you when it does the actual swapping. This enables you to get control at the critical times.

Here's how the customizable swapping feature works. Normally when you create a thread, the Threads Package assigns default swapping and context-preserving functions to the thread. If you want to use all these defaults, just call the InNewThread routine to launch a thread. To use customizable swapping, you create the thread object yourself, customize it, and then launch it. Note that you must always be sure to call the corresponding default routine from within your custom routine.

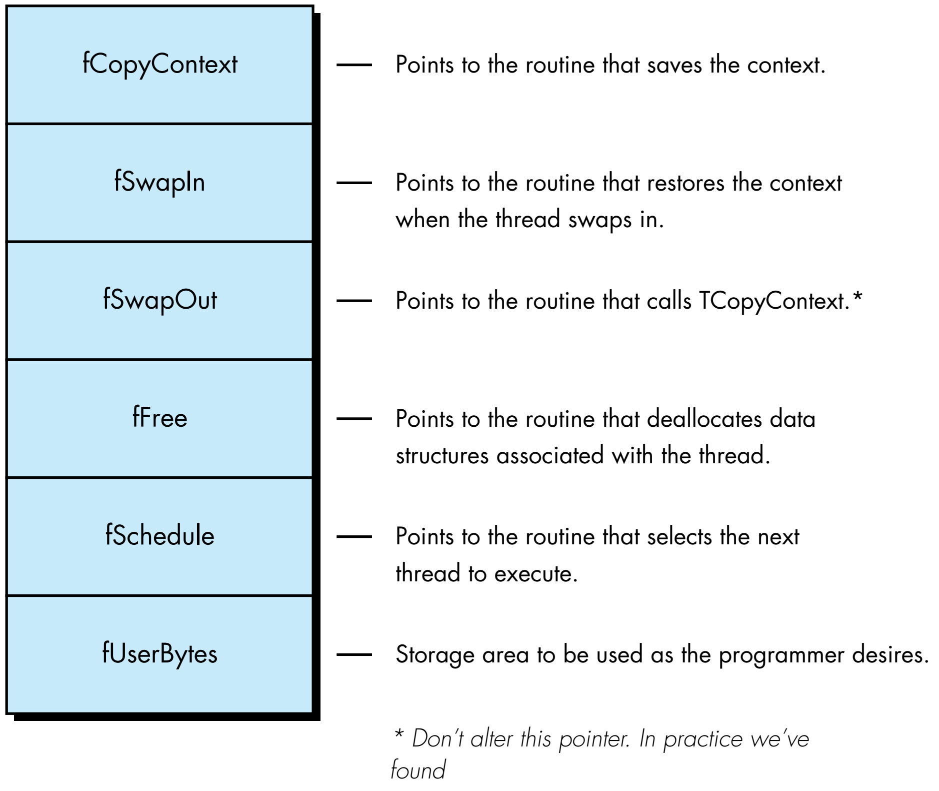

fCopyContext
Points to the routine that saves the context.
fSwapIn
Points to the routine that restores the context when the thread swaps in.
fSwapOut
Points to the routine that calls TCopyContext.*
fFree
Points to the routine that deallocates data structures associated with the thread.
fSchedule
Points to the routine that selects the next thread to execute.
fUserBytes
Storage area to be used as the programmer desires.
* Don't alter this pointer. In practice we've found

Figure 1
Customizable Routines in the Thread Structure

Remember, you don't necessarily have to use this customizable swapping technique to juggle all of your global variables. Some globals are really fixed values and don't change when your program switches threads. You only have to worry about the globals that other threads are going to change.

The following sample program demonstrates how to customize the swapping behavior of threads. Notice that there are now two ball-drawing threads. They manage to use the same global variable, gBallSize, to draw balls of different sizes. If we assume that this global is used by the NewBall procedure to determine the size of the ball, and that you don't have control over the implementation of NewBall, then you must have a way to juggle the global's value. This example shows you how to do just that:

```
pascal void MyCopyContext(ThreadHandle theThread)
{
    (**theThread).fUserBytes[0] = gBallSize ;
    TCopyContext(theThread);
}

pascal void MySwapIn(ThreadHandle theThread)
{
    gBallSize = (**theThread).fUserBytes[0] ;
    TSwapIn(theThread);
}

main()
{
    ThreadHandle ballThread;
    ThreadHandle mainThread;

    Initialize();

/* Create and customize the main thread. InitThreads will start it. */

    mainThread = NewThread(kDefaultStackSize);
    (**mainThread).fCopyContext = &MyCopyContext;
    (**mainThread).fSwapIn = &MySwapIn;
    InitThreads(mainThread, false);

/* Create, customize, and start the ball thread. */

    ballThread = NewThread(kDefaultStackSize);
    (**ballThread).fCopyContext = &MyCopyContext;
    (**ballThread).fSwapIn = &MySwapIn;
    StartThread(ballThread);
    if (InThread(ballThread))
        {
        long i;
        gBallSize = 100;
        for (i=0; i<100; i++)
            {
            NewBall();
            Yield();
            }
        EndThread(ballThread);
        }
/* Here's the main event loop. */

   gBallSize = 20;
   do
      {
      Yield();
      NewBall();
      } while (!Button());


/* This call to ExitThreads waits for all threads to die before
allowing the program to terminate. */

   ExitThreads();
}
```

Note that this example uses procedure pointers. As always with procedure pointers, make sure that they're A5 relative so that they can be dereferenced from another segment. In this case, the Threads Package will be calling your procedures at the critical moments before swapping in and swapping out. My preferred technique for ensuring that procedure pointers are A5 relative is to put the procedure in its own segment, separate from the routine that's generating the reference to it.

Figure 2 illustrates how we've customized the thread for the sample program above.

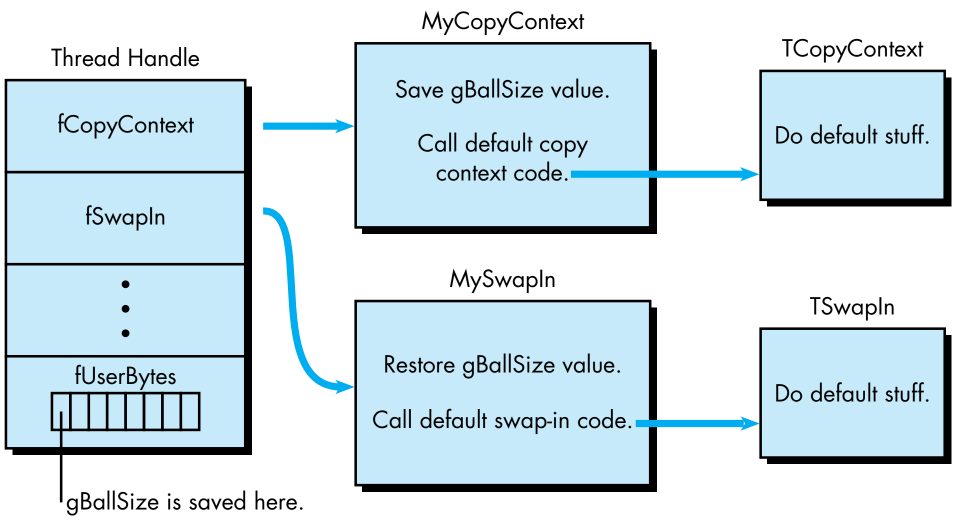


Figure 2
Customizing a Thread

TOOLBOX USE OF MEMORY BETWEEN THE STACK AND THE HEAP

Most threads implementations involve keeping a separate stack in the heap for each thread. They do their context swapping by altering the stack pointer and the stack base; the data on the stack never moves. Unfortunately, there are some routines in the Macintosh Toolbox that assume the stack remains in the same place, not in the heap.

One of the primary design goals of the Threads Package was Toolbox compatibility, so here's the solution we chose. It's a given that there is only one stack and all threads must share the use of this stack. However, since a thread needs to maintain its unique stack data and protect it from being clobbered by other threads, each thread needs to keep this data safe when it doesn't have control of the stack. The way a thread does this is by creating its own unique storage area in the heap. The Threads Package's context-swapping strategy moves data between the stack and the heap with the BlockMove instruction. As a thread swaps out, its context is moved to the heap. As a thread swaps in, its context is moved from the heap into the application's stack area.

The context-swapping code is written in such a way that interrupts can function as usual, and of course you can call Toolbox routines as usual. The heap storage associated with a thread's stack can and will grow dynamically as necessary, since it's free to move around in memory while it's not running.

Swap time using this strategy is 500 microseconds for a stack size of 256 bytes running on an SE/30. Your mileage may vary.

You must be careful not to pass pointers to stack objects between threads, since such pointers are not valid unless the associated thread is swapped in. One subtle way that this problem occurs is in the use of parameter blocks associated with asynchronous I/O. Such parameter blocks should not be allocated on the stack because the I/O operation may complete when the wrong thread is swapped in.

SEGMENT UNLOADING

When you write threaded programs for the Macintosh, you must never unload a code segment unless you're certain that there is no thread that has entered that code segment and has not yet left. In some cases, you can be sure that there's no way for a thread to yield control while it's in a specific code segment. For example, if you have some code that does some computation that stands on its own, you can be reasonably certain that there's no way for it to call other code that could result in a Yield. In cases like this, it's safe to unload the segment as usual.

We looked at several mechanisms for overcoming this problem and found that the most promising design involves unloading segments at GrowZone time. Here's how this could work: The system could call the GrowZone routine when you need more memory in the current heap zone. Since the whole idea behind unloading code segments is to free up memory, we thought that this would be a good place to

THREADS IN USE TODAY

Threads are currently in use in a product called Virtual User (APDA #M0987LL/A). This program uses a single machine, acting as the "user," to run software tests on many CPUs at once.

Virtual User used to wait until each test was done before starting something else on another test machine. That was slow, because all the testing machines were waiting for one of their siblings to finish something before getting anything to work on. Now, with threads, the controlling machine is happily juggling separate conversations with all of the testing machines simultaneously. The result is a dramatic boost in performance.

unload segments. The trick is to make sure that your GrowZone routine only unloads segments that are not needed by any thread. To ensure this, you could augment the thread structure to include linked-list pointers that would allow your custom GrowZone procedure to traverse a list of all threads (even sleeping threads) in one pass. During the traversal, GrowZone would scan the stack of each thread, looking for anything resembling a return address. If it found a return address, the associated code segment would be "needed." When all the stacks were scanned, GrowZone would simply unload all of the unneeded code segments.

THE THREADS API

Here's a description of all the routine and data structures provided by the Threads Package.

THE THREAD STRUCTURE

The API functions all access a thread through its handle. The thread structure as it's defined in the Threads.h file is as follows:

```
struct Thread
{
   struct Thing   fThing;        // Linked-list stuff.
   ThreadType     fType;         // Obsolete.
   ThreadState    fState;        // Running,pending,blocked,sleeping,ended.
   Boolean        fLocked;       // Obsolete.
   Handle         fStack;        // The storage for the stack data.
   ThreadProc     fCopyContext;  // Copy current context and store in fStack.
   ThreadProc     fSwapIn;       // Called to context-swap a thread in.
   ThreadProc     fSwapOut;      // Calls fSchedule, then fSwapIn on the nextThread.
   ThreadProc     fFree;         // Called to dispose of the thread.
   ScheduleProc   fSchedule;     // Queue this thread (if necessary), return the next one.
   long           fUserBytes[8]; // For user use.
};
```

INITIALIZING THE THREADS PACKAGE

```
pascal void   InitThreads(ThreadHandle mainThread, Boolean usesFPU);
```

This routine initializes the Threads Package. The first parameter is the handle of the main thread, which has been customized with specific swapping behavior. If you don't need customized swapping behavior for the main thread, pass nil. The second parameter indicates whether you want to swap floating-point registers. If you pass a value of true, they'll be swapped. Of course, the Threads Package is smart enough to know that some machines don't support FPUs, in which case it ignores a value of true.

CUSTOMIZING THREADS

```
pascal ThreadHandle  NewThread(long stackSize);
```

Each thread structure has a number of fields that are procedure pointers. The Threads Package assigns default procedures to these fields when it creates a thread. You can create a custom thread by calling NewThread and changing the values of the procedure pointers before giving the thread a chance to run.

Here's a list of the procedure pointers that you can change in the thread structure:

```
ThreadProc                  fCopyContext;
ThreadProc                  fSwapIn;
ThreadProc                  fSwapOut;
ThreadProc                  fFree;
ScheduleProc                fSchedule;
```

When you change one of these procedure pointers in the thread structure, you're overriding the default behavior of a given thread. You will usually customize fCopyContext and fSwapIn to save and restore globals at the appropriate moments. If you need to deallocate data structures associated with the thread, you should override fFree, which is called when the thread dies.

If you're using the default behavior, don't forget to call the corresponding default procedure appropriately within your procedure. Here's a list of the default procedures:

```
pascal void                 TCopyContext(ThreadHandle);
pascal void                 TSwapIn(ThreadHandle);
pascal void                 TSwapOut(ThreadHandle);
pascal void                 TFree(ThreadHandle);
pascal ThreadHandle         TSchedule(ThreadHandle);
```

There is a handy place to store information in the thread structure, called fUserBytes. If you store handles there, be sure to deallocate them in your override of fFree.

```
pascal void          StartThread(ThreadHandle theThread);
pascal Boolean       InThread(ThreadHandle theThread);
```

Once you've created the thread with the call to NewThread and have customized it, you call StartThread, which clones the current stack and saves it in the newly created thread structure. The call to StartThread is typically followed by a call to InThread, which returns true if the specified thread is currently running. This call is embedded in an IF statement that you use to route the respective threads. The original thread jumps over the code in the IF statement, while the new thread enters this body of code.

CONVENIENCE ROUTINES

```
pascal Boolean       InNewThread(ThreadHandle* theThread,
                                     long stackSize);
```

The InNewThread function combines the features of NewThread, StartThread, and InThread. What's different about InNewThread is that it automatically launches a thread with the default swapping behavior and doesn't give you the opportunity to customize the thread. InNewThread returns a Boolean as does InThread, and returns a thread handle in the theThread parameter. You must supply a value for stackSize, which is the number of bytes initially allocated for this thread's stack. If the number you supply is too small, the Threads Package will automatically grow the block of memory that contains the stack. Nice, huh? So if you don't know or care what stack size you need, just pass in 0.

```
pascal ThreadHandle  Spawn(ThreadHandle theThread,
                        pascal void (*threadProc)(ThreadHandle,
long),
                        long stackSize,
                        long refCon);
```

The Spawn routine is for mutants who don't like fork semantics. You supply a thread handle, or nil if you want an uncustomized thread. You also supply a procedure pointer that points to a procedure containing code for the new thread to run. The new thread dies when it returns from your procedure. You also specify a stackSize and a refCon, which allows you to pass some context information to the new thread. The refCon field is usually a pointer or a handle to a memory block that contains parameters you want to pass in.

The distinguishing characteristic of spawn semantics is that the code for the new thread is separated from the code for the original thread. Some people are more comfortable when these things are separated, but passing parameters to initialize the new thread is more work. With fork semantics, all of your local variables are right there on the stack. You don't need to package them up in a record as you do with spawn semantics.

OTHER STUFF

```
pascal ThreadHandle    GetCurrentThread();
```

The GetCurrentThread function returns the handle to the currently executing thread.

```
pascal void            Yield();
```

The Yield function is called to explicitly give control to other threads. Yield is called implicitly through other routines like Sleep. (If the current thread is going to sleep, it had better yield control to a waking thread.)

STATES OF CONSCIOUSNESS

```
pascal void            Sleep(ThreadHandle theThread);
pascal void            Wake(ThreadHandle theThread);
pascal void            EndThread(ThreadHandle theThread);
```

These routines allow you to alter a thread's state of consciousness. To put a thread to sleep, you simply call Sleep and pass it a thread handle. Usually, a thread will call Sleep to put itself to sleep, although there are some cases where this will be done by another thread. To wake a thread up, call Wake. To kill the thread, use EndThread.

THE THREADS ADVANTAGE

The Threads Package provides a nearly painless way for you to implement multiple threads of execution in your programs. All you need to learn is a handful of routines and a slightly new way of thinking about program execution. And you can gain a lot: easier, more intuitive program design; vastly simpler code; possible performance boosts; and, of course, that holy grail of Macintosh programmers, increased user satisfaction. It's a deal that's hard to refuse.

ACKNOWLEDGMENTS

I would like to thank Joe MacDougald for his Herculean contributions to the design and implementation of the Threads Package. Without his devoted effort, the Threads Package would not exist in its current form. Thanks to Tom Dowdy for the FPU register-swapping feature, and numerous other improvements. Thanks also to ATG researcher Jed Harris, who originally suggested threads as a solution to our problems. Jed helped a great deal with design issues, and some gnarly assembly code debugging.

The swapping strategy that allows the Threads Package to be Macintosh Toolbox–compatible was suggested by Donn Denman. Thanks to P. Nagarajan, the first threads user. He dropped threads into his code virtually overnight, giving us valuable input that made it possible to steer the design and implementation.

Tom Saulpaugh made significant contributions to the current design of semaphores. Thanks, Tom. Thanks to Dave Harrison for reviewing an early version of the source code for threads. Thanks to Mitchell Gass for documenting an earlier version of the Threads Package. And thanks to my mentor Larry Tesler for supporting the development of the first version of threads, and suggesting the convenience functions.

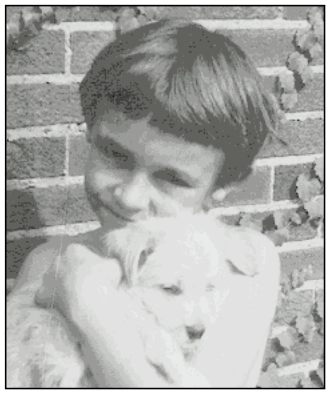
Thanks x 10^6 to my editor Geta Carlson. We had a blast working together on this article, although we've never met in person. Thanks to Paul Snively for polishing the article and championing threads in DTS. Greg Anderson, C. K. Haun, Dave Johnson, and Dave Williams all contributed valuable suggestions that were incorporated. Thanks to Monica Meffert, Louella Pizzuti, and Caroline Rose for making the article happen.

Finally, thanks to my managers Dave Leffler and Ron Metzker for putting up with me while I worked on this, and for supporting what this is leading up to.

Thanks to Our Technical Reviewers
C. K. Haun, Paul Snively, Dave Williams •

QUICKDRAW'S COPYBITS PROCEDURE: BETTER THAN EVER IN SYSTEM 7.0

With System 7.0 comes a major revision of QuickDraw. The CopyBits procedure, QuickDraw's image-processing workhorse, has had some bugs fixed and some features added. This article gives a brief overview of changes to QuickDraw and then brings you up to speed on changes to CopyBits.



KONSTANTIN OTHMER

We're seeing some impressive examples of computer image processing at the movies these days. Special effects in movies such as *Star Trek III*, *Willow*, *Back to the Future*, *Arachnophobia*, *Ghost*, and *The Abyss* were either assisted or completely generated by computer. While QuickDraw™ does not have the built-in ability to perform the highly specialized image processing necessary to produce the types of effects seen in those movies, producing such effects is not beyond the capability of the Macintosh. You can write custom routines to perform such operations as rotation, warping, and advanced filtering.

Before you get to that advanced stuff, though, you need to be familiar with QuickDraw's basic image-processing capabilities, which provide the starting point for an effects toolbox. With QuickDraw's CopyBits procedure, you can perform several standard image-processing operations, such as resizing (by stretching, shrinking, or clipping the image), colorizing, or changing the pixel depth.

CopyBits is better than ever in System 7.0. Improvements to transfer operations and colorizing mean enhanced results. And it's now easier to use a search procedure to alter colors. We'll look at these improvements in detail and will see samples of CopyBits in action after a brief overview of how QuickDraw has evolved.

KONSTANTIN OTHMER is a wild man. He whips out books, *develop* articles, and ski vacations in less time than it takes most of us to find our keys. We're not sure what position he plays on the soccer field—maybe it's "guy with the ball." He is, however, a team player. He works on QuickDraw in the system software group and helps people out all over the place. The kind of music he likes is from famous bands you haven't yet heard of. The baby picture here is just a trade show disguise; if you want a hint about what Konstantin really looks like, check out the Berlin Wall illustration in this article—if you look carefully you'll find him and Bruce Leak peeking out at you.•

A BRIEF HISTORY OF QUICKDRAW

There have been a number of QuickDraw versions since the introduction of the Macintosh in 1984. Table 1 summarizes the major QuickDraw versions. Many minor revisions and bug fixes have also occurred along the way, of course.

Table 1
A Summary of Major QuickDraw Versions

Date	**Version**	**Where Documented**
January 1984	Original B&W QuickDraw (Macintosh 128K)	*Inside Macintosh* Volume I
January 1986	B&W QuickDraw (Macintosh Plus)	*Inside Macintosh* Volume IV
March 1987	Color QuickDraw B&W QuickDraw (Macintosh II)	*Inside Macintosh* Volume V
May 1989	32-Bit QuickDraw v. 1.0	*Inside Macintosh* Volume VI
September 1989	32-Bit QuickDraw v. 1.1 (System 6.0.4, Macintosh IIci, IIfx, IIsi, and LC)	*Inside Macintosh* Volume VI
March 1990	32-Bit QuickDraw v. 1.2 (System 6.0.5)	*Inside Macintosh* Volume VI
April 1991	Color QuickDraw B&W QuickDraw (System 7.0)	*Inside Macintosh* Volume VI

Note: QuickDraw is revised for system releases and, in the past, major revisions have coincided with hardware releases. In the future, it's likely that major system releases will be independent of hardware releases.

The version of black-and-white QuickDraw that accompanied the Macintosh Plus system added the SeedFill, CalcMask, and CopyMask calls. The Macintosh II revision introduced Color QuickDraw (which supported indexed devices only) and revised the existing black-and-white QuickDraw (which is still used on 68000-based machines) to display pictures (data of type 'PICT') created in the color version.

Version 1.0 of 32-Bit QuickDraw, released as an INIT at the Developers Conference in 1989, added direct-color capability to QuickDraw. No black-and-white QuickDraw update was provided. Version 1.1 of 32-Bit QuickDraw is in ROM on the Macintosh IIci, IIfx, IIsi, and LC. Version 1.2 of 32-Bit QuickDraw, released as an INIT with System 6.0.5 and patched by the system on machines that have version 1.1 in ROM, added the OpenCPicture call and the capability of recording font names into pictures.

The System 7.0 version of Color QuickDraw integrates the functionality of 32-Bit QuickDraw into all Color QuickDraw machines and adds a variety of new features and bug fixes. In addition, System 7.0 has a new version of black-and-white QuickDraw that includes some of Color QuickDraw's functionality. (See "QuickDraw Features New in System 7.0" on the next page for more information.)

ABOUT COPYBITS

The CopyBits procedure, along with the CopyMask and CopyDeepMask calls, is the core of QuickDraw's image-processing capability. CopyBits transfers a bit image from one bitmap to another and clips the result to a specified area. With CopyBits you can perform such image-processing operations as resizing (by stretching, shrinking, or clipping the image), colorizing, and changing the pixel depth. You can use it to display on-screen the contents of an off-screen buffer.

In the System 7.0 version of QuickDraw, as in previous versions, the CopyBits procedure is defined as

```
PROCEDURE  CopyBits (srcBits,dstBits: BitMap;srcRect,dstRect: Rect;
                     mode: INTEGER; maskRgn: RgnHandle);
```

In the original black-and-white QuickDraw, CopyBits used six explicit parameters (srcBits, dstBits, srcRect, dstRect, mode, and maskRgn) and one global variable (thePort). The introduction of Color QuickDraw required an additional global variable, theGDevice, which is used to determine color information for the destination.

Although the number of variables used by CopyBits hasn't changed from earlier QuickDraw versions, several things *have* changed:

- The way transfer operations specified by the mode parameter are performed has changed to make their results predictable regardless of whether the destination device uses indexed or direct color.

- The way the notCopy transfer operation is performed has changed to improve the quality of color inversions.

QUICKDRAW FEATURES NEW IN SYSTEM 7.0

NEW IN COLOR QUICKDRAW

Custom drawing for specific screen depths. The DeviceLoop call lets applications do custom drawing for specific screen depths rather than having QuickDraw do the color translation. (If QuickDraw does the translation, a picture of a color wheel may turn out solid black when drawn on a black-and-white screen.) With DeviceLoop, you pass a drawing region, flags, and a pointer to a callback procedure that DeviceLoop will call for each different device that intersects the drawing region.

Picture Utilities. The Picture Utilities package ('PACK' 15) provides an easy way to profile the contents of a picture. It can tell you which fonts are used inside a picture so you can warn the user if one of the fonts is not available. It can also calculate the optimal color table or palette (using a predefined color pick method, or you can write your own) for displaying the picture.

Any bit depth for a mask. Before System 7.0, CopyMask's mask parameter could be only 1 bit deep. This caused the mask to be used very much like a region, selecting whether or not to copy a specific source pixel. In 7.0, the mask can be any bit depth. It specifies a blending value for merging the source and destination: black selects the source, white selects the destination, and gray provides a blend between the source and destination. Color masks can be used to blend only specific color components.

New version of the CopyMask call. The new CopyDeepMask call is an extension of CopyMask that includes a mode parameter and a region parameter. CopyDeepMask enables a blend of the source and destination to be applied to the destination using any transfer mode (not just srcCopy). Like previous versions of CopyMask, CopyMask and CopyDeepMask calls are not saved in pictures and do not print in System 7.0. (The resulting image can be printed, of course!) This may change in a future version.

NEW IN BLACK-AND-WHITE QUICKDRAW

New calls. The calls RGBForeColor, RGBBackColor, GetForeColor, GetBackColor, and QDError are now available in B&W QuickDraw.

Font names in pictures. Font names, rather than just font IDs (which may be different on different machines), are recorded into pictures, as in 32-Bit QuickDraw v. 1.2.

Custom drawing for specific screen depths. The DeviceLoop call, as described for Color QuickDraw, exists on all 7.0 machines, but because B&W QuickDraw supports only one screen device, the call is trivial.

Native resolution. OpenCPicture enables you to specify a picture's native resolution. This makes it easy to create pictures with resolutions other than 72 dpi. This feature was first available in 32-Bit QuickDraw v. 1.2.

Picture Utilities. See description for Color QuickDraw. In B&W QuickDraw, the Picture Utilities will not return a palette when you request color information.

Version 2 pictures. B&W QuickDraw previously could display version 2 pictures created on color machines, but could create only version 1 pictures. In 7.0, pictures created with OpenCPicture are version 2.

Display of 16- and 32-bit PICTs. Before System 7.0, B&W QuickDraw could display only PICTs containing indexed pixMap data; in 7.0, it can display pictures containing direct-color data.

1-bit GWorlds. In 7.0, 1-bit GWorlds are available in B&W QuickDraw. You must access the data with GetGWorldPixMap. You cannot dereference the GWorldPtr directly. On black-and-white machines, GetGWorldPixMap returns a handle to an extended bitmap (only 1 bit is supported), rather than a pixMap. You can then call GetPixBaseAddr to access the pixels.

- Dithering has been extended to improve the quality of images resulting from depth conversion, color mapping, or resizing.

- The way colorizing is performed has changed to make the results predictable for all pixel depths.

- The use of search procedures has been extended and now provides an easier mechanism for altering colors.

In the following sections we'll take a closer look at each of these improvements. We'll then watch CopyBits in action as we stretch and colorize a gray ramp, and perform RGB and CMY color separations.

IMPROVEMENTS TO TRANSFER OPERATIONS

The appearance of the result of the CopyBits procedure is determined by the mode parameter. This parameter specifies which source transfer mode is to be used and whether or not dithering should occur during transfer operations. Improvements to CopyBits in System 7.0 make the results of transfer operations independent of whether the destination device uses indexed or direct color. The new CopyBits also improves the results of color inversions and extends the use of dithering.

RESULTS INDEPENDENT OF DESTINATION DEVICE

Before System 7.0, the transfer mode specified in CopyBits' mode parameter was implemented directly by one of eight transfer operations: Copy, Or, Xor, Bic, notCopy, notOr, notXor, and notBic. For each bit in the source bitmap to be drawn, QuickDraw found the corresponding bit in the destination bitmap, performed the transfer operation on the pair of bits, and stored the resulting bit into the bit image.

This method extended naturally to the use of indexed devices in Color QuickDraw. But with the introduction of 32-Bit QuickDraw, which supported both indexed and direct-color devices, the results of the Or, Bic, and Xor transfer operations became dependent on the type of destination device. Using the Or operation with direct color—where 0 represents black and $FF represents white—resulted in pixels that went toward white, while using the Or operation on indexed pixels—where indexes typically range from 0 (white) to $FF (black)—had a result that went toward black. Bic and Xor had similar problems.

For example, many applications use the srcXor transfer mode—defined in *Inside Macintosh* Volume I as inverting destination pixels that are black in the source—when dragging a selection. In the original Color QuickDraw, this operation was performed correctly. In 32-Bit QuickDraw, on the other hand, destination pixels that were *white* in the source were inverted on direct-color devices.

In the new Color QuickDraw, the transfer modes srcOr, srcBic, and srcXor are still undefined for color pixel values, but behave correctly—that is, as documented in *Inside Macintosh* Volume I—with respect to black and white regardless of whether the destination device uses indexed or direct color. The way these modes work now as compared to the way they worked in 32-Bit QuickDraw version 1.0 for direct sources copied to a direct-color device is shown in Figure 1.

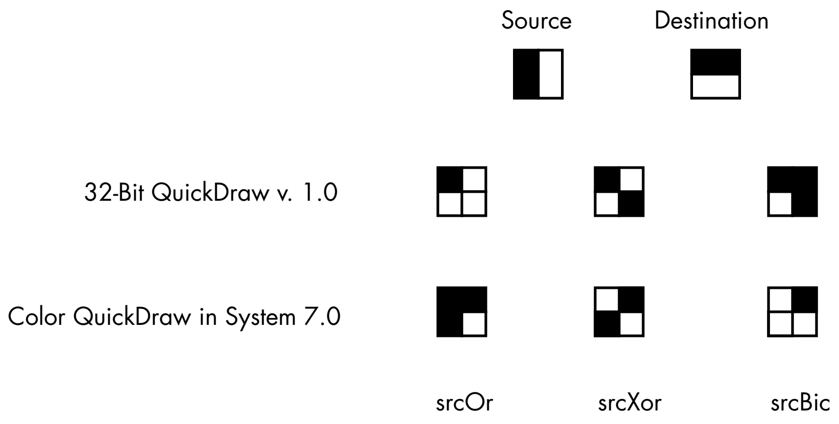


Figure 1
Results of Transfer Modes for Direct Source to Direct Destination

For all devices now, the srcOr transfer mode produces a black result where the source is black. The srcXor transfer mode inverts destination pixels where the source is black. And srcBic (which stands for "bit clear" but may be easier to remember as "black is changed") produces a white result where the source is black. All three modes leave the destination pixels under the white part of the source unchanged. (Note that using these transfer modes for colored sources, while legal, does not always produce well-defined results.)

INVERSIONS IN COLOR SPACE

Before System 7.0, notCopy was performed by inverting source index values. In System 7.0, the inversion takes place in color space, giving a much more pleasing result. Note that the trade-off for higher quality in this case is reduced speed: this operation is somewhat slower than in previous versions.

Using notSrcCopy mode to highlight items when they've been selected produces good results on screens of all depths, although it suffers from gray mapping to gray.

Figure 2 shows a button ("Squishy") highlighted using notSrcCopy mode.

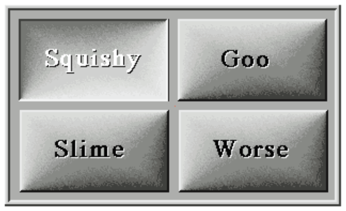


Figure 2
Button Highlighted Using notSrcCopy Mode

EXTENSIONS TO DITHERING
32-Bit QuickDraw version 1.0 introduced the dither flag. In that version of QuickDraw, setting the dither flag (bit 6 of the mode, called ditherCopy) caused dithering to occur when direct pixMaps were copied to indexed destinations.

In System 7.0, setting the dither flag in QuickDraw causes dithering to occur during any depth conversion or color mapping. For example, you can get a dither when converting an 8-bit image to a 4-bit image or a 1-bit image, or when copying between two 4-bit pixMaps that have different color tables. Figure 3 shows the effect of dithering when depth conversion occurs.

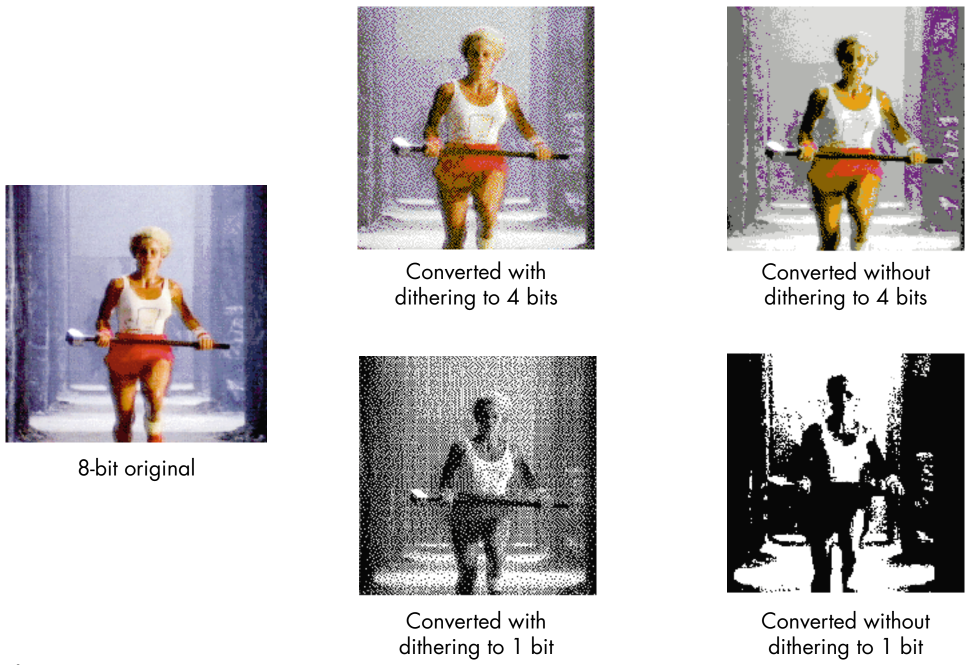
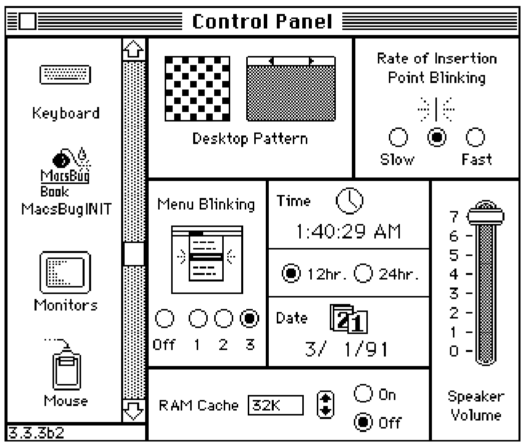
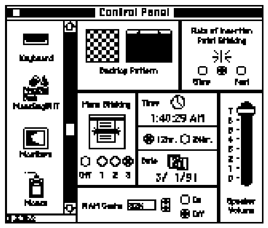

Converted with dithering to 4 bits
Converted without dithering to 4 bits
8-bit original
Converted with dithering to 1 bit
Converted without dithering to 1 bit

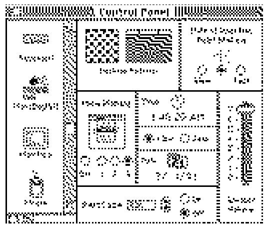
Figure 3
Depth Conversion With and Without Dithering



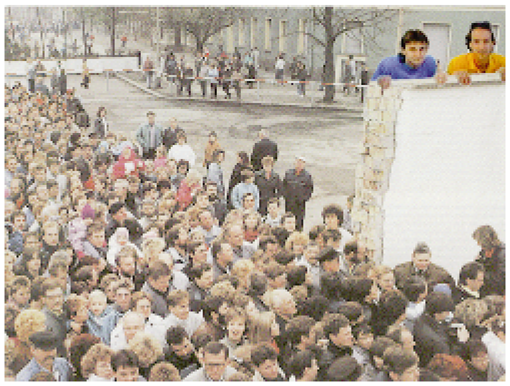
1-bit original



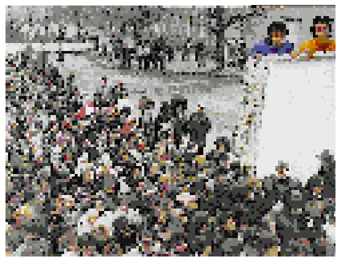
Shrunk without dithering



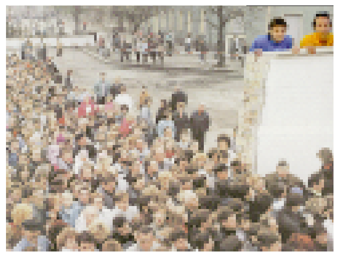
Shrunk with dithering



8-bit original



Shrunk without dithering



Shrunk with dithering

Figure 4
Resizing With and Without Dithering

In addition, setting the dither flag now affects how images are resized. In 32-Bit QuickDraw, only 32-bit pixMaps used a technique of pixel averaging in RGB space when they were shrunk. All other pixMaps were shrunk using a technique that maximizes pixel value and tends to turn shrunk pixMaps to black. In System 7.0, setting the dither flag causes pixMaps of all depths to be averaged when shrunk. Figure 4 shows the effect of dithering when shrinking a 1-bit image and an 8-bit image. Notice that the dithered result for the 1-bit image includes shades of gray as well as black and white.

Because dithering is a relatively slow process, setting the dither flag tells CopyBits that quality is more important than speed. Note, however, that direct pixMaps are always averaged when shrunk, regardless of the state of the dither flag.

IMPROVEMENTS TO COLORIZING

When CopyBits transfers an image from one bitmap to another, it refers to the foreground and background color fields of the global variable thePort. The foreground color specified there is applied to black pixels in the source and the background color is applied to white pixels. This is known as *colorizing*.

Before System 7.0, colorizing with CopyBits was performed on the indexes of the colors rather than on the color values. This meant that the results depended on the organization of the color look-up table (CLUT) of the destination GDevice. Thus, the results for multicolor images were unpredictable. This problem is illustrated in Figure 5. This was the basis for the common knowledge that the foreground and background color in the current grafPort must be set to black and white respectively or unpredictable results would occur when using CopyBits.

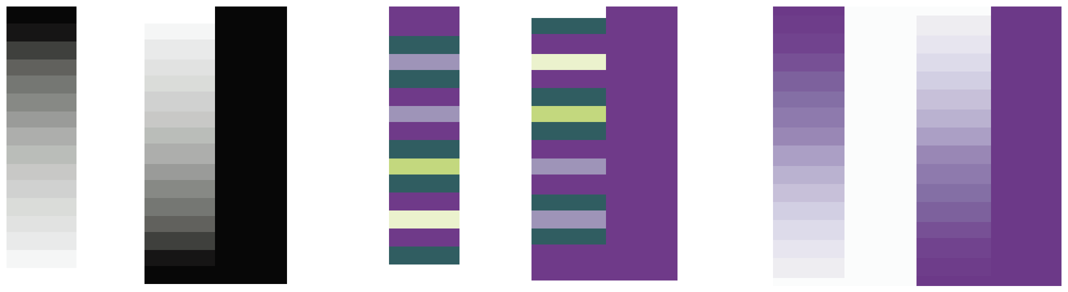

Original image
Image colorized with pre-System 7.0 CopyBits
Image colorized with System 7.0 CopyBits

Figure 5
Colorizing

In System 7.0, colorizing occurs in color space, not index space. Thus, colorizing now works as predictably for deep source pixMaps as it always has for 1-bit source pixMaps.

Colorizing is logically the last step in the CopyBits procedure. It modifies the destination pixel color as follows: all bits that are off in the source pixel are given the value of the corresponding bit in the foreground color, and all bits that are on are given the value of the corresponding bit in the background color. This is illustrated for a 16-bit pixel in Figure 6. For a foreground color of black (all components 0) and a background color of white (all components $FFFF) this operation does not change the pixel color value. The formula that performs this operation (in color space) is

result = (src AND bkColor) OR ((not src) AND fgColor)

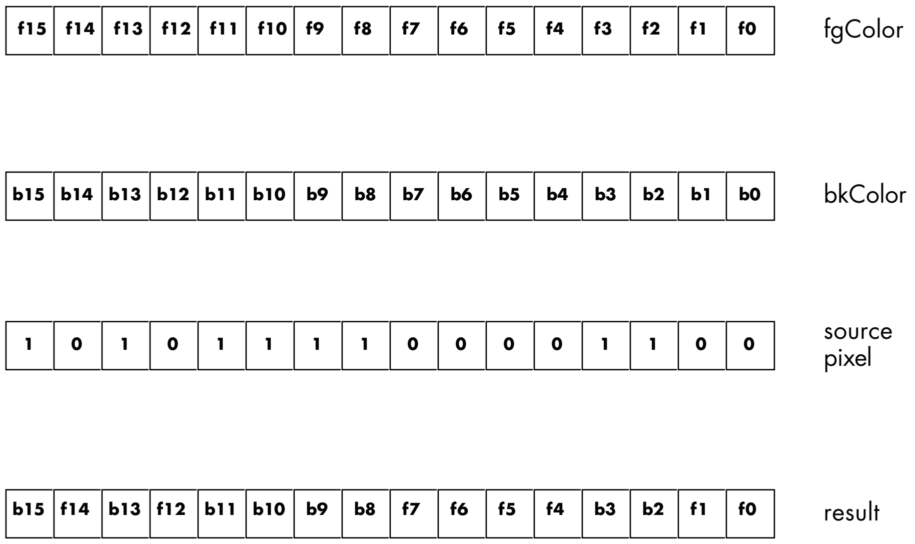

f15 f14 f13 f12 f11 f10 f9 f8 f7 f6 f5 f4 f3 f2 f1 f0
fgColor
b15 b14 b13 b12 b11 b10 b9 b8 b7 b6 b5 b4 b3 b2 b1 b0
bkColor
1 0 1 0 1 1 1 1 0 0 0 0 1 1 0 0
source pixel
b15 f14 b13 f12 b11 b10 b9 b8 f7 f6 f5 f4 b3 b2 f1 f0
result

Figure 6
How Colorizing Works in System 7.0

This operation may seem convoluted at first, but it turns out to be quite useful. For example, you can invert an image by changing the foreground color to white and the background color to black. Figure 7 shows some of the variations on one image that can be obtained simply by changing the foreground and background colors. The code samples later in this article use CopyBits colorizing to perform CMY and RGB color separation.

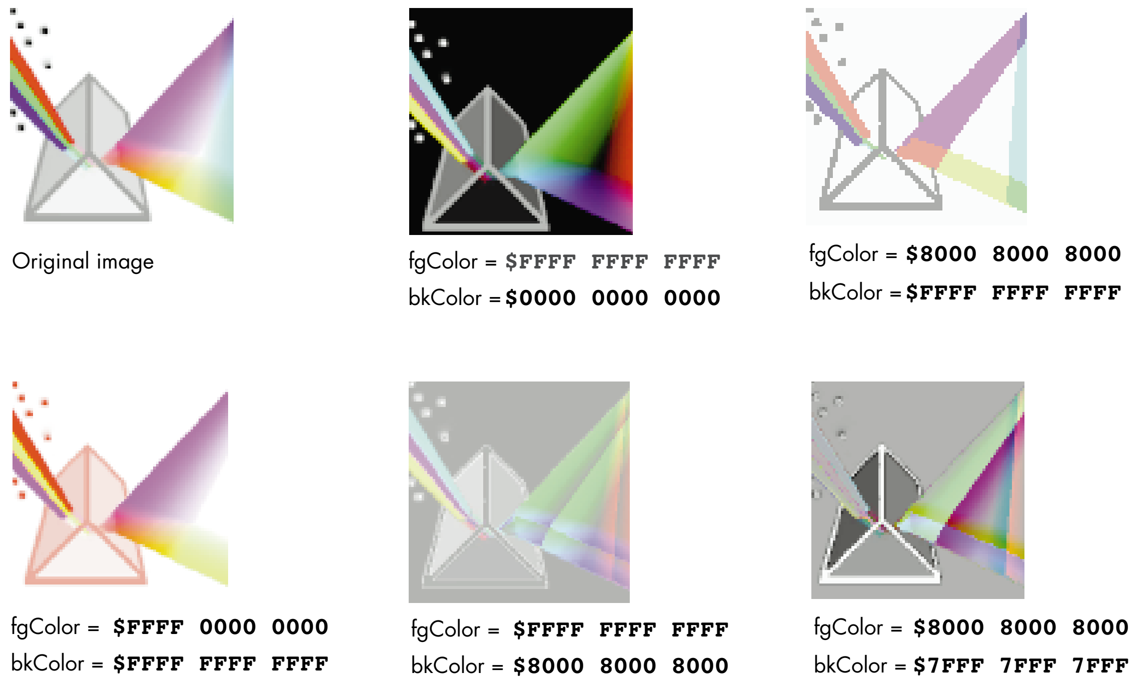

Original image
fgColor = $FFFF FFFF FFFF
bkColor = $0000 0000 0000
fgColor = $8000 8000 8000
bkColor = $FFFF FFFF FFFF
fgColor = $FFFF 0000 0000
bkColor = $FFFF FFFF FFFF
fgColor = $FFFF FFFF FFFF
bkColor = $8000 8000 8000
fgColor = $8000 8000 8000
bkColor = $7FFF 7FFF 7FFF

Figure 7
Colorized Versions of the 32-Bit QuickDraw Icon

AN EASIER WAY TO ALTER COLORS

In Color QuickDraw, destination color information comes from the current GDevice. You can attach a search procedure to a GDevice to determine how colors will appear on that device.

Before System 7.0, search procedures were used only when the source and destination pixMaps had different depths or color tables. In System 7.0, search procedures can be used in any case—for example, to alter the colors of a pixMap. In addition, the RGBColor parameter that the search procedure receives is now always a VAR parameter. (This was not true in 32-Bit QuickDraw for direct-color destinations.)

In System 7.0, a search procedure is defined as

```
FUNCTION SearchProc(VAR rgb: RGBColor; VAR position: LongInt) : Boolean;
```

On entry, the RGBColor parameter contains the color QuickDraw is trying to represent on the current device. The search procedure can do one of three things:

- It can return the index or the direct-color value (depending on the device) in the position parameter and a result of TRUE. In this case, QuickDraw draws using the color returned in the position parameter.
- It can modify the RGBColor parameter and return a result of FALSE. In this case, QuickDraw ignores the position parameter and uses its default color look-up mechanism on the returned color to find the value to draw with. For indexed devices, QuickDraw uses an inverse look-up table (ILUT) to determine which index to represent a given color with. For direct-color devices, QuickDraw merely truncates each component of the RGBColor parameter to the desired size: 5 bits for 16-bit color, 8 bits for 32-bit color.
- It can leave the RGBColor parameter unchanged, return FALSE, and still let QuickDraw do the job using the default algorithm above.

Using a search procedure in this way provides an easy mechanism for altering colors. For example, to darken an image you simply attach a search procedure that reduces the RGBColor parameter to a GDevice and then call CopyBits with that device as the current GDevice.

COPYBITS IN ACTION

The following code samples show how to do some useful things with the improved CopyBits found in Color QuickDraw in System 7.0. Example 1 shows how to stretch and colorize a gray ramp. Although the example is trivial, a number of pitfalls associated with directly accessing a GWorld's pixMap are addressed.

Example 2 shows how to do RGB and CMY color separation with CopyBits, and how to expand the source picture by a factor of 1.5. It's fairly easy to do RGB and CMY color separation using CopyBits with the correct foreground and background colors. Note that CMYK color separation (which removes gray components before separating the cyan, magenta, and yellow) is generally more useful than the simple CMY separation performed here. CMYK color separation is usually accomplished by using a search procedure.

EXAMPLE 1: STRETCHING AND COLORIZING A GRAY RAMP

The goal in this first example is to produce a red-scale ramp that fills the current window. The code merely allocates a one-pixel-wide gray-scale line and then uses CopyBits colorizing to stretch this line to the size of the window.

The first thing the code does is allocate a 32-bit off-screen GWorld to hold the one-pixel-wide line. If the allocation fails, the routine does nothing.

Next, GetGWorldPixMap is used to get a handle to the GWorld's pixMap. Note that this call did not work in pre-System 7.0 versions of QuickDraw. In those versions you could get the pixMap handle directly from the GWorld. On black-and-white QuickDraw machines, you must use GetGWorldPixMap. Note that on these machines you get the functional equivalent of a pixMap as far as GWorlds are concerned, but you do not get a true PixMapHandle.

The code then locks the pixels. This is necessary since CopyBits can move memory. Here's what we've got so far:

```
void
DoColorizedCopyBits()
{
   Rect             srcRect;
   long             * bitsPtr;
   short            iii;
   long             jjj;
   RGBColor         myrgb, savergb;
   GDHandle         oldGD;
   GWorldPtr        oldGW;
   GWorldPtr        myOffGWorld;
   PixMapHandle     myPixMapHandle;
   unsigned short   myRowBytes;
   char             mode;

   SetRect( &srcRect, 0, 0, 1, 256 );     /* Left, top, right, bottom. */
   if( NewGWorld( &myOffGWorld, 32, &srcRect, 0, 0, 0 ) == noErr)
   {
      myPixMapHandle = GetGWorldPixMap( myOffGWorld );  /* 7.0 only. */
/*    myPixMapHandle = myOffGWorld->portPixMap;        pre-7.0. */
      LockPixels( myPixMapHandle );
```

Next the code gets the base address of the pixels using the GetPixBaseAddr call. This call returns a base address that's good in 32-bit addressing mode, so the code saves the current mode and switches to 32-bit addressing mode. This is necessary to support accelerators that might keep the GWorld data cached on a card requiring 32-bit addressing. See "About 32-Bit Addressing" on the next page for more information.

ABOUT 32-BIT ADDRESSING

If your application needs to directly access the memory in a GWorld, you need to know some things about 32-bit addressing.

A tour through slot space. Slot space, and thus video memory, is at the top of the memory map, as shown in Figure 8, and sometimes requires 32-bit addressing.

In 24-bit mode, slot space ranges from $900000 to $EFFFFF, with 1 MB per slot ($s00000 to $sFFFFF where s = slot number $9 to $E). In 32-bit mode, slot space ranges from $F9000000 to $FEFFFFFF, with 16 MB per slot ($Fs000000 to $FsFFFFFF where s = $9 to $E). Super slot space, accessible only in 32-bit mode, ranges from $90000000 to $EFFFFFFF, with 256 MB per slot ($s0000000 to $sFFFFFFF where s = $9 to $E).

QuickDraw versions before 32-bit QuickDraw always use 24-bit slot space. But 24-bit slot space doesn't permit access to more than 1 MB of video memory, easily outgrown with 32-bit-per-pixel displays. Thus, video cards with more than 1 MB of video memory must be addressed in 32-bit mode. 32-Bit QuickDraw always accesses the screen in 32-bit mode, using either the 32-bit slot space or the super slot space baseAddr as given by the video ROM.

Let's look at some examples of how cards use slot space if 32-Bit QuickDraw is running:

- The original Macintosh High-Resolution Video Card uses 24-bit slot space, with a baseAddr of $Fss00000. In 24-bit mode, the stripped address is $s00000, which maps to slot s in 24-bit slot space. That address also works with 32-Bit QuickDraw because if it's used in 32-bit mode, it happens to map to 32-bit slot space as well.
- The 8•24 card uses $Fs000000 (32-bit slot space) with 32-Bit QuickDraw and $Fss00000 (24-bit slot space) with earlier versions. The 8•24 GC card uses $s0000000 (super slot space!) with 32-Bit QuickDraw and $Fss00000 with earlier versions.

32-Bit QuickDraw correctly handles pixMaps it creates—that is, pixMaps belonging to GDevices in the DeviceList and to GWorlds. However, if you create your own pixMap with your own baseAddr, the address is assumed to be good in 24-bit mode. If you pass QuickDraw a 32-bit base address, you must explicitly indicate that the address is 32-bit by setting bit 2 of the pixMap's pmVersion field.

The plot thickens. The issue of 24-bit versus 32-bit addressing becomes important when you use the GWorld calls to create a GWorld and then access the GWorld's pixels directly. To get the baseAddr of such a pixMap, you should call GetPixBaseAddr. This call returns a baseAddr that's good for certain cards only in 32-bit mode. Thus, you should always assume that the address is 32-bit and that you have to call SwapMMUMode.

If you forget to switch to 32-bit mode by calling SwapMMUMode, you've got problems. But the bug will not appear until you use an 8•24 GC card with a 2 MB DRAM upgrade kit or any other card that implements GWorlds.

Thus, to access the data at the address returned by GetPixBaseAddr you must switch to 32-bit mode with SwapMMUMode, call StripAddress on any handle that you dereference, and switch back to the original mode when you've finished accessing the pixels. Example 1 in this article shows how to correctly access a GWorld's pixels. Note that you can't make any other system calls after you've switched from 24- to 32-bit mode, since calls expect to be made in whatever mode the Macintosh was started up in.

The upshot. To access the pixels of an off-screen GWorld in System 7.0, call GetPixBaseAddr and switch to 32-bit mode. And test your application with an accelerator card that implements GWorlds. If you don't want your GWorlds to go out on a card, you can set the keepLocal flag in NewGWorld—but then you won't get the benefits of graphics acceleration.

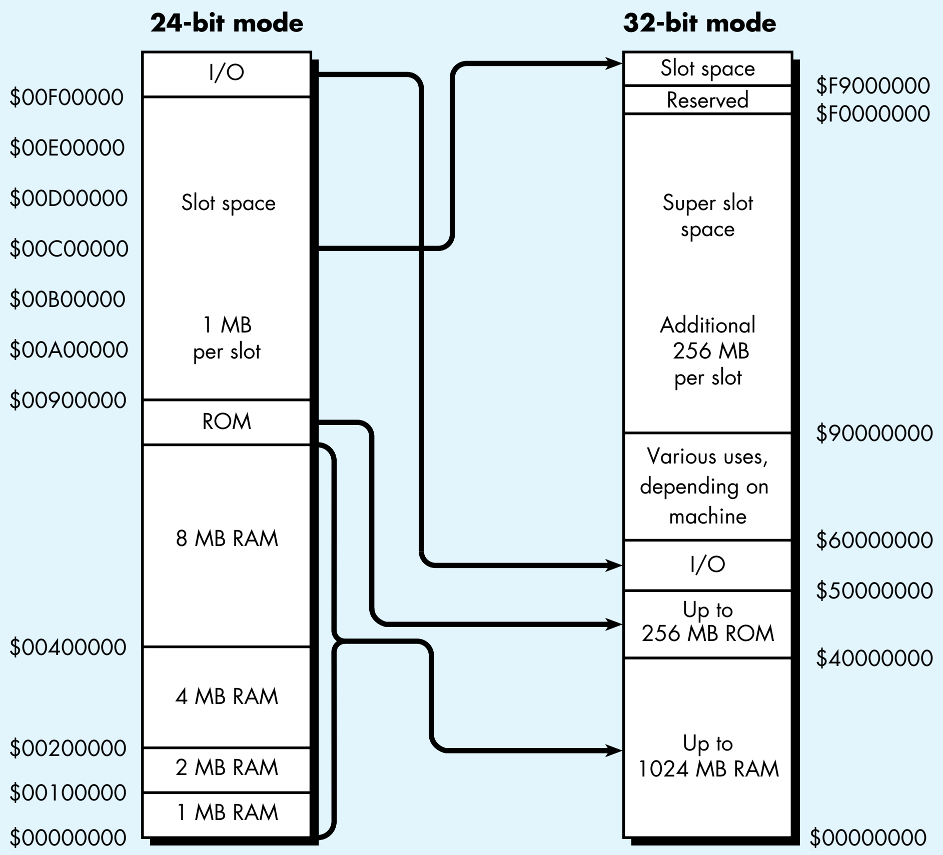


Figure 8
Macintosh II Memory Map in 24-Bit and 32-Bit Mode

```
/* Get baseAddr good in 32-bit mode. */
      bitsPtr = (long *) GetPixBaseAddr( myPixMapHandle );
      myRowBytes = (**myPixMapHandle).rowBytes & 0x3fff;
      mode = true32b;            /* Switch to 32-bit mode. */

/* Go to 32-bit addressing mode to access pixels. */
      SwapMMUMode( &mode );
```

Then the code fills the GWorld with a gray ramp. Note that you cannot make other system calls after you switch the addressing mode, since system calls expect to be made in the addressing mode the machine was booted in.

```
       for( jjj = 256-1; jjj >= 0; jjj-- )
       {
          *bitsPtr = jjj | (jjj<<8) | (jjj<<16);
          bitsPtr = (long *)((char *)bitsPtr + myRowBytes);
       }
```

Next the code switches back to the prior addressing mode, sets the foreground color to red, and uses CopyBits to stretch the line to the size of the current port and colorize it to red. Finally, the foreground color is restored and the GWorld is disposed of.

```
/* Back to old addressing. */
       SwapMMUMode( &mode );
       GetForeColor( &savergb );
       myrgb.red = 0xFFFF;
       myrgb.green = 0;
       myrgb.blue = 0;
       RGBForeColor( &myrgb );
       CopyBits( *myPixMapHandle, &thePort->portBits, &srcRect,
          &thePort->portRect, srcCopy, 0 );
       RGBForeColor( &savergb );
       UnlockPixels( myPixMapHandle );
       DisposeGWorld( myOffGWorld );
   }
}
```

EXAMPLE 2: DOING RGB AND CMY COLOR SEPARATIONS AND SCALING A SOURCE PICTURE

In addition to doing RGB and CMY color separation, the following code expands the source picture by a factor of 1.5. When QuickDraw stretches an image, it simply replicates pixel values. Thus, if you scale an image up by a factor of 3 in both the horizontal and vertical dimensions, each pixel appears in nine places in the result. But if you scale an image by a factor of 1.5, only every other pixel is repeated, so source pixels do not contribute equally to the result.

Fortunately, this problem is easy to rectify. Since CopyBits averages when shrinking with the ditherCopy flag set, you can first scale the image up by a factor of 3 and then shrink it by a factor of 2. It's easiest to visualize this process by thinking of the horizontal and vertical dimensions independently. In the vertical direction, each source pixel is first expanded to three destination pixels. Then, when the image is shrunk by a factor of 2, CopyBits averages two scanlines to produce each pixel of the result. The outcome is that each source pixel contributes equally to the result.

The following code sample produces CMY color separations that are scaled by 1.5. The first section of code draws the picture into a GWorld three times the size of the picture's bounding box.

```
void
CMYColorSeparation()
{
   Rect              dstRect;
   long              * bitsPtr;
   RGBColor          myrgb, savergb;
   GDHandle          oldGD;
   GWorldPtr         oldGW;
   GWorldPtr         myOffGWorld;
   PixMapHandle      myPixMapHandle;

   Rect              bounds;
   PicHandle         myPicHandle;

   #define           PICTResID     1000

   myPicHandle = GetPicture( PICTResID );
   if( !myPicHandle )
      return;                  /* Failed -> exit. */
   bounds = (*myPicHandle)->picFrame;
   /* Home the rect (top, left at 0, 0). */
   OffsetRect(&bounds, -bounds.left, -bounds.top);
   dstRect = bounds;
   dstRect.right *=1.5;         /* Final image = 1.5 times size of src image. */
   dstRect.bottom *=1.5;
   OffsetRect( &dstRect, 20, 20 );

   bounds.right *=3;            /* Expand by factor of 3. */
   bounds.bottom *=3;

   if( NewGWorld( &myOffGWorld, 32, &bounds, 0, 0, 0 ) == noErr)
   {
      GetGWorld(&oldGW,&oldGD);
      GetForeColor( &savergb );
      SetGWorld(myOffGWorld,nil);

      EraseRect( &bounds );     /* Clear the GWorld. */

      myPixMapHandle = GetGWorldPixMap( myOffGWorld );  /* 7.0 only*/
/*    myPixMapHandle = myOffGWorld->portPixMap;  pre-7.0. */
      LockPixels( myPixMapHandle );
      DrawPicture( myPicHandle, &bounds );
```

The GWorld now contains the picture blown up three times in both directions. Next it's copied four times to the window to a dstRect 1.5 times the size of the original picture. The first three times, the GWorld is color-separated to yellow, magenta, and cyan; then the original image is drawn.

```
       SetGWorld(oldGW,oldGD);             /* Copy to window. */

/* Get the yellow component. */
       myrgb.red = 0xFFFF;
       myrgb.green = 0xFFFF;
       myrgb.blue = 0;
       RGBForeColor( &myrgb );
       CopyBits( *myPixMapHandle, &thePort->portBits, &bounds, &dstRect,
          ditherCopy + srcCopy, 0 );
       OffsetRect( &dstRect, 220, 0 );

/* Get the magenta component. */
       myrgb.red = 0xFFFF;
       myrgb.green = 0;
       myrgb.blue = 0xFFFF;
       RGBForeColor( &myrgb );
       CopyBits( *myPixMapHandle, &thePort->portBits, &bounds, &dstRect,
          ditherCopy + srcCopy, 0 );
       OffsetRect( &dstRect, -220, 220 );

/* Get the cyan component. */
       myrgb.red = 0;
       myrgb.green = 0xFFFF;
       myrgb.blue = 0xFFFF;
       RGBForeColor( &myrgb );
       CopyBits( *myPixMapHandle, &thePort->portBits, &bounds, &dstrect,
          ditherCopy + srcCopy, 0 );
       OffsetRect( &dstRect, 220, 0 );

/* Copy original image. */
       myrgb.red = 0;
       myrgb.green = 0;
       myrgb.blue = 0;
       RGBForeColor( &myrgb );
       CopyBits( *myPixMapHandle, &thePort->portBits, &bounds, &dstRect,
          ditherCopy + srcCopy, 0 );
       RGBForeColor( &savergb );
       UnlockPixels( myPixMapHandle );
       DisposeGWorld( myOffGWorld );
    }
}
```

Getting the RGB components is similar. Simply replace the previous four CopyBits calls with the following:

```
/* Get the red component. */
       myrgb.red = 0;
       myrgb.green = 0;
       myrgb.blue = 0;
       RGBForeColor( &myrgb );
       myrgb.red = 0xFFFF;
       myrgb.green = 0;
       myrgb.blue = 0;
       RGBBackColor( &myrgb );
       CopyBits( *myPixMapHandle, &thePort->portBits, &bounds, &dstRect,
         ditherCopy + srcCopy, 0 );
       OffsetRect( &dstRect, 220, 0 );

/* Get the green component. */
       myrgb.red = 0;
       myrgb.green = 0xFFFF;
       myrgb.blue = 0;
       RGBBackColor( &myrgb );
       CopyBits( *myPixMapHandle, &thePort->portBits, &bounds, &dstRect,
         ditherCopy + srcCopy, 0 );
       OffsetRect( &dstRect, -220, 220 );

/* Get the blue component. */
       myrgb.red = 0;
       myrgb.green = 0;
       myrgb.blue = 0xFFFF;
       RGBBackColor( &myrgb );
       CopyBits( *myPixMapHandle, &thePort->portBits, &bounds, &dstRect,
         ditherCopy + srcCopy, 0 );
       OffsetRect( &dstRect, 220, 0 );

/* Original. */
       myrgb.red = 0xffff;
       myrgb.green = 0xffff;
       myrgb.blue = 0xffff;
       RGBBackColor( &myrgb );
       CopyBits( *myPixMapHandle, &thePort->portBits, &bounds, &dstRect,
         ditherCopy + srcCopy, 0 );
```

The result of these color separations is shown in Figure 9.

FOR MORE INFORMATION

The usual sources (*Inside Macintosh,* Tech Notes) will soon be augmented by a new QuickDraw book by David Surovell, Frederick Hall, and Konstantin Othmer in the Macintosh Inside Out series from Addison-Wesley. *Debugging Macintosh Software With Macsbug* by Konstantin Othmer and Jim Straus (Addison-Wesley, 1991) contains a great deal of information on debugging QuickDraw-related (as well as other) problems. •

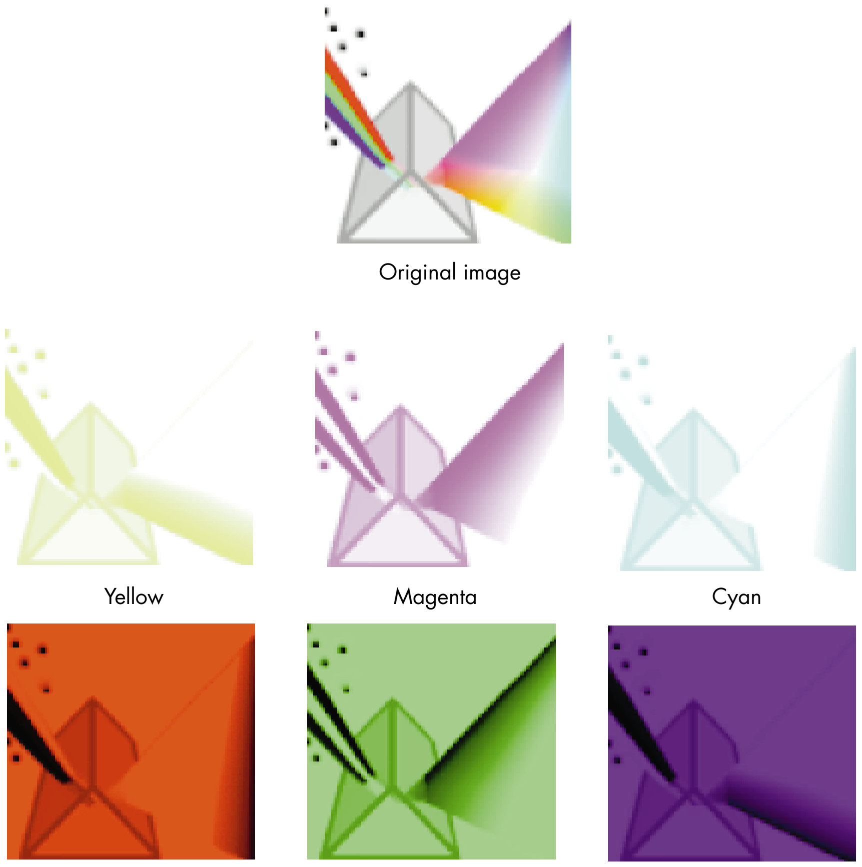


Figure 9
CMY and RGB Color Separations Generated Using CopyBits

LATER, DUDE

CopyBits is the workhorse at the core of QuickDraw's image-processing capabilities. As you've learned in this article, it's better than ever in System 7.0. CopyBits transfer operations now give higher-quality images and produce reliable results for all pixel depths and regardless of whether the destination device uses indexed or direct color. The GDevice's search procedure provides an easy way to alter colors. Color separations have become fairly easy to do. Can Hollywood-style special effects be far behind?

Thanks to Our Technical Reviewers
Rich Biasi, Jean-Charles Mourey, Guillermo Ortiz, Forrest Tanaka •

THE LOW-DOWN ON IMAGE COMPRESSION
BY FORREST TANAKA

As you know, when pixel data is included in a PICT, the data is usually packed. Pixel maps that are 8 or fewer bits deep pack fairly well using straight run-length encoding of bytes (that is, the PackBits routine), but compressing direct pixels using run-length encoding doesn't work very well. Here's what QuickDraw does with direct pixels in PICTs:

If the packType field contains 1, no compression is done at all. The complete pixel image is saved in the PICT. If the packType field contains 2 and the pixel map is 32 bits per pixel, all that's done is that the alpha channel byte is removed. So this

```
00 FF FF FF  00 FF FF FF
```

is compressed to this

```
FF FF FF  FF FF FF
```

If the packType field contains 3 and the pixel map is 16 bits per pixel, run-length encoding is done, but not through PackBits. Instead, a run-length encoding algorithm private to QuickDraw is used. This algorithm is very similar to PackBits, but where PackBits compresses runs of bytes, this routine compresses runs of words. The format of the resulting data is exactly the same as described in Technical Note #171, Things You Wanted to Know About _PackBits, but you'll get words instead. For example, let's say the 16-bit pixel image begins with these pixel values:

```
AAAA AAAA AAAA 0000 2A2A AAAA AAAA AAAA
AAAA F0F0 0101 2A2A 4F4F AAAA AAAA AAAA
```

After being packed by QuickDraw's internal compression routine, this becomes

```
FEAA AA01 0000 2A2A FDAA AA03 F0F0 0101
*      *            *      *
2A2A 4F4F FEAA AA
          *
```

where the asterisks mark the flag counter bytes. Notice that you can't assume the pixel values are word-aligned. PackBits packs data 127 bytes at a time, for up to 32,767 total bytes; similarly, the internal compression routine packs data 127 words at a time.

If the packType field contains 4 and the pixel map is 32 bits per pixel, run-length encoding via PackBits is done, but only after some preprocessing. QuickDraw first rearranges the color components of the pixels so that each color component of every pixel is consecutive. So the following three pixels

```
00 FF FF FF  00 FF C0 00  00 FF 80 00
a0 r0 g0 b0  a1 r1 g1 b1  a2 r2 g2 b2
```

are rearranged to become

```
FF FF FF  FF C0 80  FF 00 00
r0 r1 r2  g0 g1 g2  b0 b1 b2
```

In the row below the pixel values a = alpha channel, r = red, g = green, b = blue, and the number is the pixel offset. The first three bytes are the red components of the three pixels, the next three bytes indicate the green components of the three pixels, and so on. The alpha channel isn't included unless the cmpCount field contains 4 rather than the normal 3. If cmpCount contains 4, all the alpha channel bytes are placed before the red bytes. Once this is done, PackBits is called to compress the rearranged data.

These are the only four compression schemes (including no compression) that are supported for direct pixel maps in PICTs. As always, reading PICTs yourself puts you in danger of not being able to read PICTs generated by future versions of QuickDraw. However, for compatibility reasons, these compression algorithms as described here probably won't change in the future. It's possible that new values for packType could be implemented, though.

Luke speaks

PRINT HINTS FROM LUKE & ZZ

COLOR PRINTING WITH LASERWRITER 6.0 REVISITED

With the release of 32-Bit QuickDraw version 1.0, Apple wanted to find a way to support color printing on the high-end (albeit black-and-white) LaserWriter® II, as well as other, third-party devices. So, the great implementors (GIs) created a new LaserWriter driver—version 6.0. This driver added a new Color/Grayscale button to the print dialog, allowing users to print their "way cool" color pictures that were created with 32-Bit QuickDraw. All was happy in the land of Apple's new color model. But wait! There was a problem lurking on the horizon, a problem called the PostScript Offending Command Error.

Imagine this scenario: You've just created a cool 32-Bit QuickDraw picture, a true masterpiece, and all seems to be going well. You decide to print your picture and show it off to Mom. You choose Print from the File menu; the print dialog appears, and you click OK. A few minutes pass, and voilà: a printed page containing your picture. Life is gooood.

With this success, you're now dreaming of other pictures that you'll be able to create and print. You create another cool 32-bit picture. You choose Print from the File menu; the print dialog appears, and you click OK. A few minutes pass, and this time the PostScript Offending Command Error dialog appears, looking something like this:

```
Error: LimitCheck; Offending Command:
080AGOBBLEDEGOOK0B
```

What's this? An offending command? But all you wanted to do was print your color picture. You were able to print your first picture a few minutes ago. What's going on? Unfortunately, LaserWriter driver version 6.0 does not reliably print images that are deeper than 8 bits. We won't bore you with the details: just think of the LaserWriter driver as a shark, ready to swallow a surfer off the California coast. But instead of the surfer dude it was expecting, in rushes the surfer dude's surfboard. The LaserWriter driver chokes on deeper images just as our poor shark chokes on the surfboard; the driver is ready to receive a particular variable but occasionally receives something different, and doesn't know what to do with it. The result: the PostScript Offending Command Error. Life is no longer so good; your dreams are beginning to fade away. But I can see a solution appearing on the horizon . . .

Actually, there are three possible solutions: you can use LaserWriter driver version 6.1 (or version 7.0 when it's available); you can depth-convert your image from 32 bits to 8 bits using 32-Bit QuickDraw; or you can use the PostScript® image operator to generate PostScript code for your image, and send it directly down to the LaserWriter. Let's look at each solution.

Using LaserWriter driver version 6.1 is the simplest solution. LaserWriter driver version 6.1 likes data of any depth, no matter when it's sent. So, if you're printing to LaserWriter driver version 6.1, life is happy when you're printing your 32-bit images—but how can you be sure that you're using LaserWriter driver version 6.1?

You can call PrDrvrVers, which is provided by the Printing Manager to enable your application to determine which version of a particular printer driver you're talking to. But there's a minor problem with this call. You don't know if you're talking to a

PETE "LUKE" ALEXANDER spends much of his Developer Technical Support time diving deep into the bowels of the Printing Manager, where he never turns up his nose at a challenge or at odiferous code that needs explaining. Although that kind of diving is fun, he prefers the balmy blue waters of anyplace (preferably far from computers) that has both beach and beer close together. If he can't get away from it all on the beach somewhere, he'll settle for getting above it all in his glider; with Luke, being up in the air about something takes on a whole new meaning. Fortunately, not everything on the horizon is blue sky. Luke's looking forward to the cool new printing architecture that will make his job (and yours) a lot easier. He's preparing for this new architecture by spending a week sailing around the Caribbean—figuring that he'd better start getting used to a life of leisure. Until that leisure can become a lifestyle, you can count on seeing lots from Luke.•

PostScript LaserWriter or some other device. So, you must dive into the bowels of the print record for additional information. You need to check the high byte of the wDev field of the TPrStl record to determine a particular driver and version. But wait! You thought checking wDev was evil. In this case, using wDev is OK because you're not checking for particular functionality of a driver, and thereby not making your code device dependent. If wDev is 3, you know that you're talking to a PostScript LaserWriter. Non-PostScript LaserWriters (for example, the LaserWriter IIsc) have a different value for the wDev field. You would then call PrDrvrVers to determine if you're talking to LaserWriter driver version 6.1. If PrDrvrVers returns 61, you know that you're using LaserWriter driver version 6.1, and life is good again. If wDev is 3, and the driver version is less than 61, you're not using LaserWriter driver version 6.1, so you have a little more work to do: try the next solution.

The next possible solution is to use 32-Bit QuickDraw to depth-convert your 32-bit image to 8 bits by using 32-Bit QuickDraw's GWorld support. You would first create an 8-bit GWorld containing a grayscale CLUT, and use CopyBits to copy your 32-bit image into it. You would then use CopyBits to copy the 8-bit image directly into the printer's grafPort, and voilà—your image would be printing. This approach works with LaserWriter driver version 6.0 and later.

Your final option is to use the PostScript image operator to generate PostScript code that represents your 32-bit image. This approach is a little more complex than the 32-Bit QuickDraw idea. To send your data down to the LaserWriter, you would need to use the PostScriptHandle PicComments with the image operator. If you're already sending PostScript code to the LaserWriter, this is probably the best approach.

By the way, if you don't have a copy of LaserWriter driver version 6.1, it's available on the *Developer Essentials* disc. If you want to ship this version of the driver with your application, you should contact Apple Software Licensing for the details.

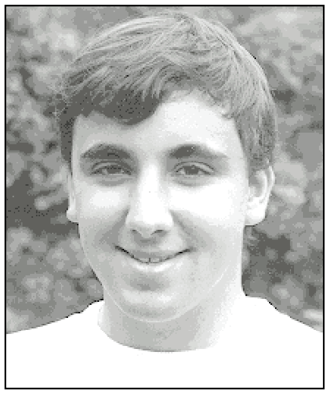
In conclusion, we have some good news and some bad news. The good news is that we've fixed the 32-bit image printing problem that was present in LaserWriter driver versions 6.0, 6.0.1, and 6.0.2. LaserWriter driver version 6.1 will allow you to print pictures that are 1 bit to 32 bits deep without any problems. The bad news is that if you want your application to print all depths of pictures with LaserWriter driver version 6.0, you're going to need to do a little extra work, either depth-converting your pictures from 32 bits to 8 bits before print time, or using the PostScript image operator to generate PostScript code for your image. Now that's not so bad, is it?

For details regarding the use of the PostScriptHandle PicComments, take a look at Technical Note #91, Optimizing for the LaserWriter—PicComments. •

For details about depth-converting your 32-bit images or using the PostScript image operator, see Technical Note #72, Color Printing. •

MACTCP COOKBOOK: CONSTRUCTING NETWORK-AWARE APPLICATIONS

The Macintosh is now a full-fledged player in the world of TCP/IP networking. MacTCP, an implementation of TCP/IP for the Macintosh, lets applications take advantage of a protocol suite that is used extensively by many makes of computers. This article attempts to demystify the process of MacTCP programming and provides a library of calls that can be used easily by anyone familiar with Macintosh programming.



STEVE FALKENBURG

TCP/IP, which stands for *Transmission Control Protocol/Internet Protocol*, was developed by the U.S. Department of Defense Advanced Research Products Agency (DARPA) and used initially on the ARPANET, a national research network created by DARPA in the late 1960s. Although the ARPANET no longer exists, the TCP/IP protocols are used on many large-scale networks. Many of these networks are interconnected and are known collectively as the *Internet.*

The TCP/IP protocol stack, shown in Figure 1, is composed of several layers. At the lowest layer, the Internet Protocol (IP) handles transmitting packets of information from one host to another. Above this network level, TCP/IP provides two transport layer protocols: Transmission Control Protocol (TCP) and User Datagram Protocol (UDP). TCP provides reliable connection-based service, while UDP is not connection based. The MacTCP® driver gives the programmer interfaces to TCP and UDP, but not to the lower-level IP. This article deals only with TCP programming. For information on MacTCP UDP programming, consult the *MacTCP Programmer's Guide.*

Several application-level protocols use TCP to provide user-level service. The Simple Mail Transfer Protocol (SMTP) is used to send electronic mail, the Network News Transfer Protocol (NNTP) is used to transfer and post news, the File Transfer Protocol (FTP) is used to transfer files between machines, and the Finger protocol is used to retrieve user information. MacTCP does not include programming interfaces or implementations for any of these application-level protocols.

STEVE FALKENBURG just started his new life as an Apple Developer Technical Support engineer, but when writing this he was still a computer engineering student at the University of Michigan at Ann Arbor. Last summer he worked in Apple's Advanced Technology Group as an intern, and he claims to have emerged from the experience totally normal (a summer in ATG may not be long enough, but a full-time job in DTS is another story altogether). He says there is absolutely nothing weird about him (which he thinks is a shame), but we're convinced we'll either unearth something or inspire it. In addition to attending classes, he's been working for the university's Computer-Aided Engineering Network, supporting and programming Macintoshes. When not working or studying, he's been seen in the stands at U of M football

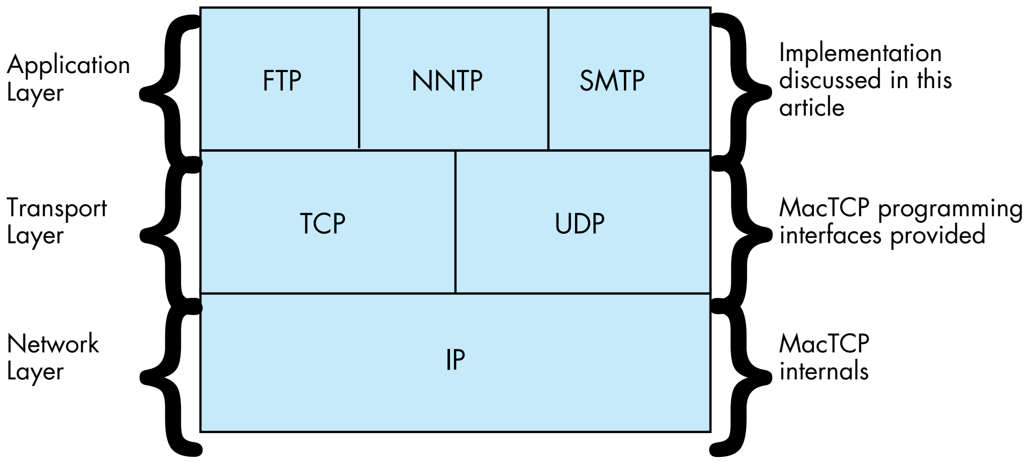


Figure 1
TCP/IP Protocol Stack

With connection-based protocols, such as TCP, a connection is defined as a bidirectional open line of communication between two hosts. Data is guaranteed to be received in the same order as it was sent, and in TCP, data reliability is ensured. To open a connection between two computers, the initiating program sends an open command containing the network address of the remote computer to MacTCP. If the remote computer is listening for a connection, it acknowledges the connection, and data can then be transferred on the connection stream. If the remote computer is not listening for a connection, the open command fails. Once all transactions have been completed, the connection may be closed by either computer.

Network addressing is essential to this process. Each device connected to a TCP/IP network is assigned a unique 4-byte address, also known as the IP number, as shown in Table 1. A unique name can also be assigned to each network entity. MacTCP provides a name service mechanism, called the Domain Name Resolver (DNR), which translates network names to addresses and vice versa.

Table 1
Network Name to Address Mapping

Network Name	IP Number
sol.engin.umich.edu	141.212.4.65
mondo.engin.umich.edu	141.212.68.14
freebie.engin.umich.edu	141.212.68.23
daneel.engin.umich.edu	141.212.68.30
maize.engin.umich.edu	141.212.10.56

games (so don't misinterpret him when he yells "Go Big Blue"), attending loud rock concerts, going to movies, and reading Cyberpunk (his favorite is William Gibson).•

In addition, there's a 2-byte port number associated with TCP connections. Each port number is usually mapped to the type of application-level service the sender is requesting. For example, the NNTP protocol always operates on TCP port 119. This mapping is shown in Table 2. MacTCP does not provide a service for translating between service name and port number.

Table 2
Service to Port Number Mapping

Service	**Port Number**
ftp	21
telnet	23
smtp	25
finger	79
nntp	119

MACTCP PROGRAMMING BASICS

This section focuses on the basics of MacTCP programming, while the remainder of the article discusses high-level, easy-to-use routines. MacTCP is a driver for the Macintosh that provides access to TCP/IP through the Device Manager. By making standard Device Manager PBControl calls, programs can access TCP/IP transport protocols to control and maintain connections. The standard TCP parameter block, defined in TCPPB.h, is shown here:

```
typedef struct TCPiopb {
  char                 fill12[12];
  TCPIOCompletionProc  ioCompletion;      /* address of completion routine */
  short                ioResult;          /* result of call (>0 = incomplete) */
  char                 *ioNamePtr;
  short                ioVRefNum;
  short                ioCRefNum;         /* MacTCP driver reference number */
  short                csCode;            /* command code */
  StreamPtr            tcpStream;         /* pointer to stream buffer */
  union {
       struct TCPCreatePB create;         /* var for TCPCreate,TCPRelease */
       struct TCPOpenPB open;             /* var for TCPActiveOpen,TCPPassiveOpen */
       struct TCPSendPB send;             /* var for TCPSend */
       struct TCPReceivePB receive;       /* var for TCPNoCopyRcv,TCPRcvBfrReturn,TCPRcv */
       struct TCPClosePB close;           /* var for TCPClose */
       struct TCPAbortPB abort;           /* var for TCPAbort */
       struct TCPStatusPB status;         /* var for TCPStatus */
       struct TCPGlobalInfoPB globalInfo; /* var for TCPGlobalInfo */
       } csParam;
} TCPiopb;
```

From the start of the parameter block up to the tcpStream parameter, this structure is a standard Device Manager control block. Three fields must be filled in:

- csCode, which specifies the driver command to be executed.
- ioCRefNum, which contains the MacTCP driver reference number.
- tcpStream, which contains a pointer to the pertinent TCP stream. This stream pointer, which is described in more detail later, is the unique identifier for a connection.

If the call is made asynchronously, as all MacTCP calls may be, a pointer to a completion routine can be specified in ioCompletion. Depending on the type of call made, there are various ways to fill in the union at the end of this parameter block.

TCP DRIVER CALLS
A description of some of the common commands and their parameter blocks follows. Unless otherwise specified, a value of zero for any parameter indicates the default value.

TCPCreate (csCode = 30)
TCPRelease (csCode = 42)

```
typedef struct TCPCreatePB {
   Ptr            rcvBuff;         /* pointer to area allocated for stream buffer */
   unsigned long  rcvBuffLen;      /* length of stream buffer */
   TCPNotifyProc  notifyProc;      /* address of asynchronous notification routine */
   Ptr            userDataPtr;
}TCPCreatePB;
```

TCPCreate allocates a MacTCP stream to be used for opening or listening for a connection. The rcvBuff parameter should be a pointer to a block of memory previously allocated as a stream buffer; set rcvBuffLen to the length of that buffer. If you want to be interrupted when the connection state changes, set notifyProc to the address of an asynchronous notification routine (ASR). The code in this article doesn't make use of the ASR feature, so in this case notifyProc should be set to nil. A pointer to the created stream is returned in tcpStream.

TCPRelease removes the stream pointed to by tcpStream from all MacTCP-internal stream lists. It returns a pointer to the stream buffer in rcvBuff. When TCPRelease completes successfully, this buffer should be disposed of by the calling program.

TCPPassiveOpen (csCode = 31)
TCPActiveOpen (csCode = 32)

```
typedef struct TCPOpenPB {
   byte ulpTimeoutValue;     /* upper-layer protocol timeout */
   byte ulpTimeoutAction;    /* upper-layer protocol timeout action */
   byte validityFlags;       /* validity flags for options */
   byte commandTimeoutValue; /* timeout value for command */
   ip_addr remoteHost;       /* IP address of the remote host */
   tcp_port remotePort;      /* TCP port to connect to on the remote host */
   ip_addr localHost;        /* local IP number */
   tcp_port localPort;       /* local port from which connection originates */
   byte tosFlags;            /* type of service requested */
   byte precedence;          /* priority of connection */
   Boolean dontFrag;         /* if true, don't fragment packets */
   byte timeToLive;          /* maximum number of hops for packets */
   byte security;            /* security option byte */
   byte optionCnt;           /* number of IP options */
   byte options[40];         /* other IP options (def in RFC 894) */
   Ptr userDataPtr;
}TCPOpenPB;
```

TCPPassiveOpen listens for an incoming connection from a specific host and port. The command completes when a remote host connects to the port monitored by this command. Store the remote host address in remoteHost and specify the remote TCP port number in remotePort. Connections from any host and port are possible if these values are set to zero. Set the localPort parameter to the local TCP port number or to zero to assign an unused port. ULP ("ulp" in the parameter names) stands for *upper-layer protocol.* The ulpTimeoutValue parameter is the maximum amount of time MacTCP allows for a connection to complete after the connection process has started. If this timeout is reached, and the value of ulpTimeoutAction is zero, the ASR, if present, is called and the ULP timer is reset. When the timeout is reached, if ulpTimeoutAction is nonzero, the command returns an error. The validityFlags parameter indicates which of the other command parameters are specified explicitly. Bit 6 is set if the ULP timeout action is valid; bit 7 is set if the ULP timeout value is valid. Descriptions for the rest of the entries in this structure can be found in the *MacTCP Programmer's Guide.* In most cases, they can all be set to zero, indicating default values should be used.

TCPActiveOpen attempts to initiate a connection with a remote host and completes when this connection is established. The parameters are identical to TCPPassiveOpen, with the following exceptions: There's no command timeout parameter, although the ULP timeout is available. You must fully specify the remote host address and remote port in remoteHost and remotePort, respectively, since it's impossible to initiate a connection to an arbitrary host without direction.

TCPSend (csCode = 34)

```
typedef struct TCPSendPB {
   byte ulpTimeoutValue;           /* upper-layer protocol timeout */
   byte ulpTimeoutAction;          /* upper-layer protocol timeout action */
   byte validityFlags;             /* validity flags for options */
   Boolean pushFlag;               /* true if data should be sent immediately */
   Boolean urgentFlag;             /* identifies the data as important */
   Ptr wdsPtr;                     /* pointer to write data structure */
   unsigned long sendFree;
   unsigned short sendLength;
   Ptr userDataPtr;
}TCPSendPB;
```

TCPSend sends data to a remote host along an open connection stream. The definitions of ulpTimeoutValue, ulpTimeoutAction, and validityFlags are the same as in TCPPassiveOpen. The pushFlag parameter is set if the data should be sent immediately and the urgentFlag option can be set to indicate that the data is important. The data to be sent is stored in a write data structure (WDS). The format of this structure is simply an array of wdsEntry structures, which are length/pointer pairs. You terminate the WDS by setting the last entry's length to zero.

```
typedef struct wdsEntry {
   unsigned short length;          /* length of buffer */
   char * ptr;                     /* pointer to buffer */
} wdsEntry;
```

TCPRcv (csCode = 37)

```
typedef struct TCPReceivePB {      /* for receive and return rcv buff calls */
   byte commandTimeoutValue;       /* timeout value for command */
   byte filler;
   Boolean markFlag;               /* true if start of read data structure is urgent data */
   Boolean urgentFlag;             /* true if read data structure ends in urgent data */
   Ptr rcvBuff;                    /* pointer to data that has been received */
   unsigned short rcvBuffLen;      /* amount of data in bytes that has been received */
   Ptr rdsPtr;                     /* pointer to read data structure */
   unsigned short rdsLength;
   unsigned short secondTimeStamp;
   Ptr userDataPtr;
}TCPReceivePB;
```

TCPRcv receives incoming data on an already established connection. Allocate a buffer for the incoming data and store a pointer to this location in rcvBuff. Store the maximum length to be received in rcvBuffLen. This value is changed to the number of bytes received when the command completes. Store the timeout value for this command in commandTimeoutValue. Finally, use markFlag and urgentFlag to delimit the start and end of urgent data blocks.

TCPClose (csCode = 38)

```
typedef struct TCPClosePB {
    byte ulpTimeoutValue;      /* upper-layer protocol timeout */
    byte ulpTimeoutAction;     /* upper-layer protocol timeout action */
    byte validityFlags;        /* validity flags for options */
    Ptr userDataPtr;
}TCPClosePB;
```

TCPClose indicates to the remote host that the caller has no more data to send on the connection. It's assumed that the remote host will then issue a close command, which permits the connection to close. However, the connection will not close until both hosts issue this command. The parameters to this call are described in other calls.

TCPAbort (csCode = 39)

```
typedef struct TCPAbortPB {
    Ptr userDataPtr;
}TCPAbortPB;
```

TCPAbort closes a connection without asking the remote host for permission. This command does not ensure that all data transfers have completed. The parameters to this call are described earlier.

MEDIUM-LEVEL TCP CALLS

Since calling the Device Manager is a painful experience for some programmers, a small library of intermediate routines can speed up development time. This article includes such a library. There's one medium-level call provided for each associated TCP driver command, so these calls simply isolate programmers from filling out parameter blocks. This seems to be a big plus with most programmers. The source code for the calls is on the *Developer Essentials* disc.

Several of the medium-level calls have hooks that allow them to be called asynchronously. Any procedure containing an async flag can be called in this manner. If this is done, the parameter block used to make the call is returned in returnBlock. The calling program must then poll returnBlock->ioResult to determine when the command has completed. As with any other Device Manager call, the ioResult field remains positive while the command is executed and then changes to zero or a negative value upon completion, indicating the call's result code. Any number of calls may be awaiting completion, since medium-level routines dynamically allocate space for parameter blocks. Completion routine hooks are not provided by these routines, but could easily be added.

For a more complete reference to the TCP driver calls, please see the *MacTCP Programmer's Guide.*•

If a medium-level routine is called with the async flag false, or if the routine doesn't have an async flag, the underlying PBControl call is still called asynchronously. While awaiting completion of the command, the medium-level routines call a routine defined by

```
Boolean GiveTime(unsigned long sleepTime);
```

This callback permits the application to carry out other small tasks. In the examples in this article, GiveTime spins the cursor and calls WaitNextEvent from a secondary main event loop to handle a subset of normal program operation and to give other applications time to execute.

At this point, you may be thinking that figuring out proper values for these routines is as much of a pain as filling out parameter blocks. For this reason, another set of high-level routines is provided for sending and receiving data.

HIGH-LEVEL TCP CALLS

High-level TCP calls further simplify and generalize the process of calling MacTCP. Write data structures are not required to send data, only a single timeout value is allowed, and other TCP specifics (mark, push, and so on) have been removed. Instead, the high-level calls are a core set of routines that are both understandable and easy to use. In most cases, it's to your advantage to use these routines, since they abstract the MacTCP programming interface to resemble a generic connection-based protocol scheme. This opens the possibility of porting these high-level routines to another protocol stack, such as AppleTalk®. In fact, if the high-level routines were modified to use AppleTalk Datastream Session Protocol (ADSP), any code written using the high-level calls for MacTCP could be compiled for use on an AppleTalk network. The source code for these routines is on the *Developer Essentials* disc.

The operation of high-level MacTCP calls is fairly straightforward. For each asynchronous routine, there's a corresponding routine to call when the asynchronous command completes. The moment of completion can be determined by polling returnBlock->ioResult. This returnBlock parameter is the same as the one returned by the medium-level routines and contains the MacTCP parameter block used in the pending asynchronous call.

NAME RESOLVER CALLS

Before you can build a useful network application, you must consider name-to-address resolution. Name-to-address resolution provides a means of converting from domain names (unique string identifiers for computers on a TCP/IP network) to IP numbers (4-byte addresses) and vice versa. In general, people can remember the name of a computer (for example, goofy.apple.com) more easily than they can remember a network address (for example, 90.1.0.10). Translation tables between the names and numbers can be stored on the local machine (the MacTCP Hosts file, for example) or on a remote server. Remote access to network numbers is provided through the Domain Name Server protocol. The MacTCP Domain Name Resolver allows name/address translations using both the static table and remote server methods.

The MacTCP Developer's Kit (APDA #M0230LL/D) ships with the file dnr.c, a set of C routines providing a programming interface to the name resolver. Several important calls from this code module are described here.

```
OSErr OpenResolver(char *fileName);
```

OpenResolver initializes the name resolver. As a parameter to the call, you can supply a local file containing important hosts. Passing nil for this filename defaults to the Hosts file in the current System Folder.

```
OSErr StrToAddr(char *hostName, struct hostInfo *hostInfoPtr,
                ResultProcPtr ResultProc, char *userDataPtr);
```

StrToAddr converts a string of the form "W.X.Y.Z" or "goofy.apple.com" to its 4-byte IP address. The hostInfo struct that you pass in looks like this:

```
typedef struct hostInfo  {
  long rtnCode;
  char cname [255];
  unsigned long addr [NUM_ALT_ADDRS];
};
```

```
OSErr AddrToName(ip_addr addr, struct hostInfo *hostInfoPtr,
                ResultProcPtr ResultProc, char *userDataPtr);
```

AddrToName performs a reverse lookup to retrieve a host name, given a 4-byte IP address.

```
OSErr CloseResolver();
```

CloseResolver closes the resolver and deallocates the resources and the address cache that has accumulated.

Calls to these dnr.c routines can be combined to construct a fairly simple routine to convert host names to IP addresses:

```
/* CvtAddr.c
  Converts host names to IP numbers
  written by Steve Falkenburg
*/
```

```
#include <Types.h>
#include <MacTCPCommonTypes.h>
#include <AddressXLation.h>
#include "CvtAddr.h"

pascal void DNRResultProc(struct hostInfo *hInfoPtr,char *userDataPtr);

/*  ConvertStringToAddr is a simple call to get a host's IP number, given the name
    of the host.
*/

OSErr ConvertStringToAddr(char *name,unsigned long *netNum)
{
    struct hostInfo hInfo;
    OSErr result;
    char done = 0x00;

    if ((result = OpenResolver(nil)) == noErr) {
        result = StrToAddr(name,&hInfo,DNRResultProc,&done);
        if (result == cacheFault)
            while (!done)
                ; /* wait for cache fault resolver to be called by interrupt */
        CloseResolver();
        if ((hInfo.rtnCode == noErr) || (hInfo.rtnCode == cacheFault)) {
            *netNum = hInfo.addr[0];
            strcpy(name,hInfo.cname);
            name[strlen(name)-1] = '\0';
            return noErr;
        }
    }
    *netNum = 0;

    return result;
}

/*  This is the completion routine used for name resolver calls.
    It sets the userDataPtr flag to indicate the call has completed.
*/
pascal void DNRResultProc(struct hostInfo *hInfoPtr,char *userDataPtr)
{
#pragma unused (hInfoPtr)

    *userDataPtr = 0xff;    /* Setting the user data to nonzero means we're done. */
}
```

PROGRAMMING A SIMPLE MACTCP APPLICATION

The core set of routines provided in the earlier sections makes it fairly easy to write a small but useful TCP application. These core routines can be stacked together to form a framework for a basic MacTCP application, as shown in Figure 2. Note that no module accesses a module that's farther than one level away. This provides the programmer the flexibility needed to improve the networking library or switch protocol stacks without losing functionality.

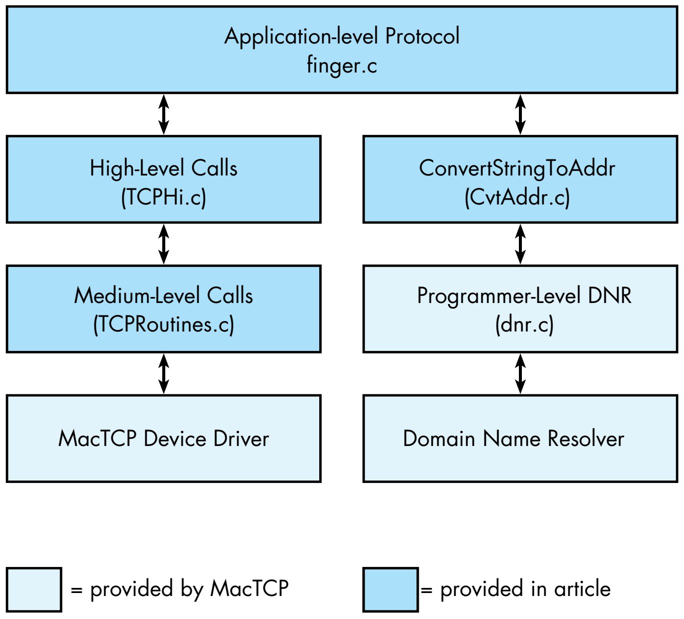


Figure 2
Finger Code Modularization

One of the simplest and most widely used network utilities is finger. Implemented on most UNIX workstations, finger is used to retrieve personal information (such as phone number and address) for a particular user. The finger utility accesses information through the Finger User Information Protocol, discussed in RFC 1194 on the *Developer Essentials* disc. The protocol operates on a client/server model. Figure 3 is a diagram of a sample transaction.

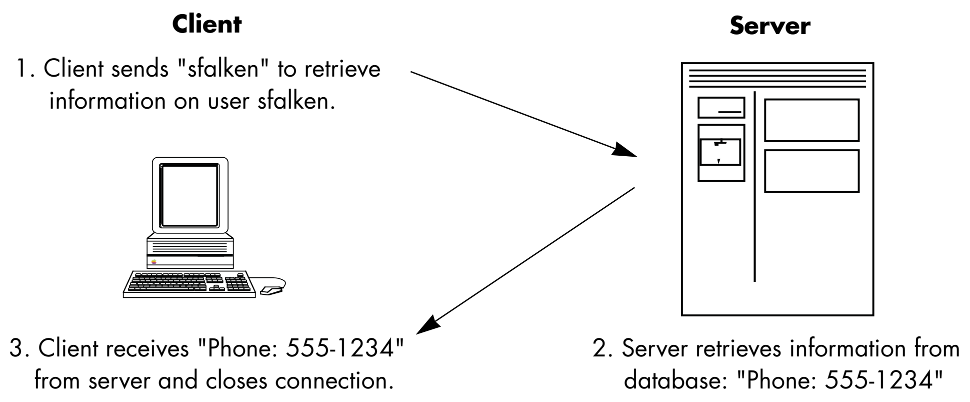

Client
1. Client sends "sfalken" to retrieve information on user sfalken.
Server
3. Client receives "Phone: 555-1234" from server and closes connection.
2. Server retrieves information from database: "Phone: 555-1234"

Figure 3
Finger Protocol Transaction

The code for a simple MPW tool to implement the Finger protocol is shown here, with accompanying explanation:

```
/* MacTCP finger client                        */
/* written by Steven Falkenburg                */
/*                                             */

#include <Types.h>
#include <Memory.h>
#include <Packages.h>
#include <CursorCtl.h>
#include <String.h>

#include "CvtAddr.h"
#include "MacTCPCommonTypes.h"
#include "TCPPB.h"
#include "TCPHi.h"

/* constants */

#define kFingerPort 79  /* TCP port assigned for finger protocol */
#define kBufSize 16384  /* Size in bytes for TCP stream buffer and receive buffer */
#define kTimeOut 20     /* Timeout for TCP commands (20 sec. pretty much arbitrary) */
```

```
/* function prototypes */

void main(int argc,char *argv[]);
OSErr Finger(char *userid,char *hostName,Handle *fingerData);
OSErr GetFingerData(unsigned long stream,Handle *fingerData);
void FixCRLF(char *data);
Boolean GiveTime(short sleepTime);

/* globals */

Boolean gCancel = false;  /* This is set to true if the user cancels an operation. */

/*      main entry point for finger                                    */
/*                                                                     */
/*      usage: finger <user>@<host>                                    */
/*                                                                     */
/*      This function parses the args from the command line,           */
/*      calls Finger() to get info, and prints the returned info.      */

void main(int argc,char *argv[])
{
   OSErr err;
   Handle theFinger;
   char userid[256],host[256];

   if (argc != 2) {
      printf("Wrong number of parameters to finger call\n");
      return;
   }

   sscanf(argv[1],"%[^@]@%s",userid,host);

   strcat(userid,"\n\r");

   err = Finger(userid,host,&theFinger);

   if (err == noErr) {
      HLock(theFinger);
      FixCRLF(*theFinger);
      printf("\n%s\n",*theFinger);
      DisposHandle(theFinger);
   }
   else
      printf("An error has occurred: %hd\n",err);
}
```

```
/*     Finger()                                                   */
/*     This function converts the host string to an IP number,    */
/*     opens a connection to the remote host on TCP port 79, sends */
/*     the id to the remote host, and waits for the information on */
/*     the receiving stream. After this information is sent, the   */
/*     connection is closed down.                                 */

OSErr Finger(char *userid,char *hostName,Handle *fingerData)
{
   OSErr err;
   unsigned long ipAddress;
   unsigned long stream;

   /* open the network driver */

   err = InitNetwork();
   if (err != noErr)
      return err;

   /* get remote machine's network number */

   err = ConvertStringToAddr(hostName,&ipAddress);
   if (err != noErr)
      return err;

   /* open a TCP stream */

   err = CreateStream(&stream,kBufSize);
   if (err != noErr)
      return err;

   err = OpenConnection(stream,ipAddress,kFingerPort,kTimeOut);
   if (err == noErr) {
      err = SendData(stream,userid,(unsigned short)strlen(userid),false);
      if (err == noErr)
         err = GetFingerData(stream,fingerData);
      CloseConnection(stream);
   }

   ReleaseStream(stream);
   return err;
}

OSErr GetFingerData(unsigned long stream,Handle *fingerData)
{
   OSErr err;
```

```
    long bufOffset = 0;
    unsigned short dataLength;
    Ptr data;

    *fingerData = NewHandle(kBufSize);
    err = MemError();
    if (err != noErr)
        return err;

    HLock(*fingerData);
    data = **fingerData;
    dataLength = kBufSize;

    do {
        err = RecvData(stream,data,&dataLength,false);
        if (err == noErr) {
            bufOffset += dataLength;
            dataLength = kBufSize;
            HUnlock(*fingerData);
            SetHandleSize(*fingerData,bufOffset+kBufSize);
            err = MemError();
            HLock(*fingerData);
            data = **fingerData + bufOffset;
        }
    } while (err == noErr);

    data[0] = '\0';

    HUnlock(*fingerData);
    if (err == connectionClosing)
        err = noErr;
}

/* FixCRLF() removes the linefeeds from a text buffer. This is */
/* necessary, since all text on the network is embedded with   */
/* carriage return linefeed pairs.                              */

void FixCRLF(char *data)
{
    register char *source,*dest;
    long length;

    length = strlen(data);

    if (*data) {
        source = dest = data;
        while ((source - data) < (length-1)) {
            if (*source == '\r')
                source++;
            *dest++ = *source++;
        }
        if (*source != '\r' && (source - data) < length)
            *dest++ = *source++;
        length = dest - data;
    }

    *dest = '\0';
}

/* This routine would normally be a callback for giving time to  */
/* background apps.                                               */

Boolean GiveTime(short sleepTime)
{
    SpinCursor(1);
    return true;
}
```

The main points in the execution of this program can be traced as follows:

1.	Get userid, host	
2.	Initialize MacTCP	`InitNetwork();`
3.	Get address of host	`ConvertStringToAddr(hostName,&ipAddress);`
4.	Make a TCP stream	`CreateStream(&stream,kBufSize);`
5.	Connect to finger host	`OpenConnection(stream,ipAddress,kFingerPort,kTimeOut);`
6.	Send userid across stream	`SendData(stream,userid,(unsigned short)strlen(userid),false);`
7.	Receive finger information from stream	`RecvData(stream,data,&dataLength,false);`
8.	Close connection	`CloseConnection(stream);`
9.	Release stream	`ReleaseStream(stream);`
10.	Quit program	

Once the host name and user ID are received, MacTCP is initialized by a call to InitNetwork. The IP number of the host is then retrieved by a call to ConvertStringToAddr. If this is successful, CreateStream creates a TCP stream, and OpenConnection opens a connection on that stream to the finger port on the remote host. Next, SendData sends the user ID along this connection. The finger information is received through repeated calls to RecvData. Once all data has been sent, CloseConnection closes the connection, and the stream is removed with

ReleaseStream. Finally, the program terminates, leaving the MacTCP driver open. Here's a sample run of the finger tool in action:

```
finger sfalken mondo.engin.umich.edu <enter>
Login name: sfalken                 In real life: Steven Falkenburg
Phone: 555-1234
Directory: /u/sfalken               Shell: /bin/csh
On since Feb 26 07:08:28 on ttyp2 from rezcop.engin.umi
No Plan.
```

The code for the finger tool can be compiled and linked with the high- and medium-level routines into an MPW tool. This example shows how a standard network protocol can be easily encapsulated into a simple program. The high-level networking calls act somewhat like a Macintosh Toolbox Manager, since they encapsulate the complexity of TCP programming. These routines can be used by any programmer with a user-level knowledge of networking, making network programming less of a mysterious art.

NEWSWATCHER—A COMPLEX TCP APPLICATION

The finger tool is limited in scope, but the same techniques can be used to construct more complex programs. NewsWatcher, a Macintosh-based network news reader, provides an example.

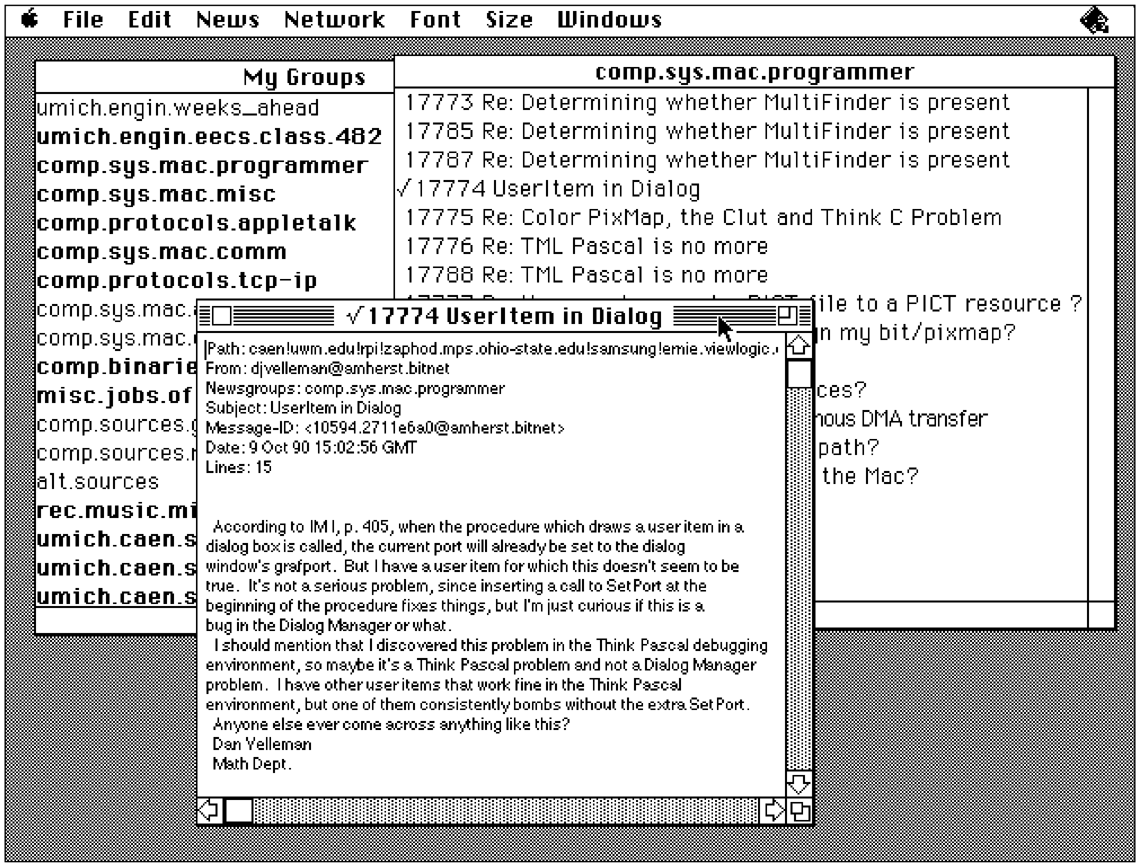
One of the most popular services available on the Internet is network news, an international forum for discussion of almost any topic you can imagine. Anyone can post and read messages. Since thousands of messages are posted every day, the messages are divided into more than a thousand newsgroups. Examples of these diverse groups include

comp.sys.mac.programmer	Macintosh programming discussions
comp.sys.mac.misc	Miscellaneous Macintosh ramblings
alt.flame.spelling	People insulting others' spelling skills
alt.alien.visitors	Discussions of alien life—intelligent or otherwise

The volume of network news bombarding readers necessitates a usable interface. I wrote NewsWatcher to provide that interface.

NAVIGATING THE NET WITH NEWSWATCHER

Before plunging into a deep technical discussion of the NewsWatcher code structure, a brief description of the interface is in order. A typical NewsWatcher screen is shown in Figure 4.



```
umich.engin.weeks_ahead
umich.engin.eecs.class.482
comp.sys.mac.programmer
comp.sys.mac.misc
comp.protocols.appletalk
comp.sys.mac.comm
comp.protocols.tcp-ip
comp.sys.mac.
comp.sys.mac.
comp.binarie
misc.jobs.of
comp.sources.
comp.sources.
alt.sources
rec.music.mi
umich.caen.s
umich.caen.s
umich.caen.s
```

```
 17773 Re: Determining whether MultiFinder is present
 17785 Re: Determining whether MultiFinder is present
 17787 Re: Determining whether MultiFinder is present
√17774 UserItem in Dialog
 17775 Re: Color PixMap, the Clut and Think C Problem
 17776 Re: TML Pascal is no more
 17788 Re: TML Pascal is no more
                                    ile to a PICT resource ?
                                    n my bit/pixmap?

                                    ces?
                                    nous DMA transfer
                                    path?
                                    the Mac?
```

```
Path: caen!uwm.edu!rpi!zaphod.mps.ohio-state.edu!samsung!emie.viewlogic.
From: djvelleman@amherst.bitnet
Newsgroups: comp.sys.mac.programmer
Subject: UserItem in Dialog
Message-ID: <10594.2711e6a0@amherst.bitnet>
Date: 9 Oct 90 15:02:56 GMT
Lines: 15


  According to IM I, p. 405, when the procedure which draws a user item in a
dialog box is called, the current port will already be set to the dialog
window's grafport.  But I have a user item for which this doesn't seem to be
true.  It's not a serious problem, since inserting a call to SetPort at the
beginning of the procedure fixes things, but I'm just curious if this is a
bug in the Dialog Manager or what.
  I should mention that I discovered this problem in the Think Pascal debugging
environment, so maybe it's a Think Pascal problem and not a Dialog Manager
problem.  I have other user items that work fine in the Think Pascal
environment, but one of them consistently bombs without the extra SetPort.
  Anyone else ever come across anything like this?
  Dan Velleman
  Math Dept.
```

Figure 4
A Typical NewsWatcher Screen

The program is based on a multiwindow browser interface, similar to the Finder (without the icons). A window containing all active newsgroups is always available, and users can get a full article subject list for a group by double-clicking that group's name. This opens up another browsing window, containing the subjects. These subjects, in turn, can be opened to display the full text of individual articles in separate windows. A user who wants to keep track of a few specific newsgroups and see only new articles in those groups can "subscribe" to the specific groups. This is done by choosing New Group Window from the File menu, which creates an empty group window. The desired groups can then be dragged from the main Newsgroups window to this custom group window. The new window can be browsed like any other group window. When the user is ready to quit, the custom group list can either be saved to disk or uploaded to a UNIX® news file (using FTP). This gives users access to their group list from multiple computers.

INTRODUCTION TO NNTP

The Network News Transfer Protocol (NNTP) is a popular protocol for transmitting and accessing network news on the Internet. This protocol, which runs out of TCP port 119 on a news host computer, is described in detail in RFC 977 on the *Developer Essentials* disc. NNTP, like finger, is based on the client-server model. An NNTP server, usually one per site, stores a copy of each newsgroup and new article. Clients contact this server to request news or post new articles.

The protocol is based on a request-response model. Clients contact the NNTP server, make a request, and receive a response. A sample session with an NNTP server is shown below. Commands sent from the client (in this case a Macintosh) are shown in italics.

```
200 srvr1.engin.umich.edu NNTP server version 1.5.10 + serve.c GNUS patch (3 October 1990)
ready at Tue Oct  9 23:46:12 1990 (posting ok).

HELP

100 This server accepts the following commands:
ARTICLE     BODY         GROUP
HEAD        LAST         LIST
NEXT        POST         QUIT
STAT        NEWGROUPS    HELP
IHAVE       NEWNEWS      SLAVE

Additionally, the following extension is supported:

XHDR        Retrieve a single header line from a range of articles.

Bugs to Stan Barber (Internet: nntp@tmc.edu; UUCP: ...!bcm!nntp)
.
GROUP comp.sys.mac.misc
211 281 3035 3744 comp.sys.mac.misc
ARTICLE 3035
220 3035 <1990Sep14.145124.25214@midway.uchicago.edu> Article retrieved; head and body
follow.

...article text here...
.
QUIT
205 srvr1.engin.umich.edu closing connection.  Goodbye.
```

Not surprisingly, the user interface code for NewsWatcher, though complex, has virtually nothing to do with MacTCP programming and is beyond the scope of this article. This code, however, is included on the *Developer Essentials* disc for those interested in a multiwindow browsing system.

NEWSWATCHER NNTP ROUTINES

The Network News Transfer Protocol (NNTP) is easily implemented from the medium-level TCP calls described in the first part of this article. (For information on NNTP, see "Introduction to NNTP.") The NNTP calls in NewsWatcher are encapsulated into a module named *NNTPLow.c.* The external functions in this module, along with a description of their purpose, are described here:

```
OSErr StartNNTP(void);
```

StartNNTP initializes the network resources needed to establish an NNTP connection with a remote server. Once all proper drivers have been initialized, this routine opens a connection to the local NNTP server, whose name is stored in the program's configuration file. This connection is maintained throughout the life span of the program.

```
OSErr CloseNewsConnection(void);
```

CloseNewsConnection gracefully terminates a connection with a remote NNTP server. This command is executed only when NewsWatcher is quitting.

```
OSErr ResetConnection(void);
```

ResetConnection is called when a communication error occurs between the client and server. This routine terminates and attempts to reestablish connection to the NNTP server.

```
OSErr GetGroupList(short *numGroups);
```

GetGroupList puts the list of newsgroups in the global variable gGroupList. To get the list of groups from the server, the command LIST is sent to the server, which responds with a list of all known newsgroups.

```
OSErr GetMessages(char *newsGroup,long first,long last,
               TSubject *subjects,long *numSubjects,char *hdrName);
```

GetMessages returns a set of subject headers specified by group name and article number range. This routine operates by sending the command XHDR *groupname first-last* to the NNTP server and parsing the response.

```
OSErr GetArticle(char *newsGroup,char *article,char **text,
                 long *length, long maxLength);
```

GetArticle retrieves the full text of the article in group newsGroup named *article*. This procedure sends the command ARTICLE *article* to the NNTP server.

These procedures are called in response to user requests for articles and subject lists. As you can see, there's absolutely no network dependence at this point, even on the NNTP protocol. It would be trivial to write a set of routines with identical function prototypes that treated a file hierarchy on a hard disk as a set of newsgroups and articles. This layered isolation approach is of critical importance in network programming, since network-level protocols may change while applications remain the same.

OTHER PROTOCOLS IN NEWSWATCHER

In addition to NNTP, several other network protocols are implemented for NewsWatcher. These protocols include the Simple Mail Transfer Protocol (SMTP) and the File Transfer Protocol (FTP).

Simple Mail Transfer Protocol (SMTP). SMTP gives users the ability to respond to article postings through electronic mail. This protocol, like NNTP, operates on a request-response stream. SMTP servers listen on TCP port 25, and the protocol is described in detail in RFC 821 on the *Developer Essentials* disc. NewsWatcher contains a single procedure, SendSMTP, that takes care of setting up a connection to the server, sending the message, and disconnecting. The function prototype for this function is as follows:

```
Boolean SendSMTP(char *text,unsigned short tLength);
```

The SendSMTP code is contained within the SMTPLow.c code module. This separation allows the code to be used in other programs easily. The routine calls functions in TCPLow.c (the medium-level routines) and is called from netstuff.c.

File Transfer Protocol (FTP). NewsWatcher includes FTP-based routines that allow users to send lists of newsgroups to, and receive them from, a file on a remote machine. This protocol uses a control stream, running on TCP port 21, to set up file transfers. When a transfer is initiated, a secondary data stream is opened on a negotiated TCP port. The file transfer is completed by means of this secondary stream, and the data stream is then closed down. For a detailed description of the protocol and command set used, see RFC 959 on the *Developer Essentials* disc. To shield programmers from the complications of FTP, I wrote several high-level routines to implement it:

```
OSErr FTPInit(ProcPtr statusCallback);
```

FTPInit initializes network resources required for FTP transfers. The single parameter provided is a pointer to a callback procedure called when a status message is received from a remote host.

```
OSErr FTPFinish(void);
```

Call this routine when file transfers are finished to release network resources used to support the FTP session.

```
OSErr FTPConnect(unsigned long *connID,char *address,char *userID,
                 char *password);
```

Call FTPConnect with the address of the remote machine, along with the user ID and password of the account needed to access that machine.

```
OSErr FTPDisconnect(unsigned long connID);
```

After all transfers to a particular host have been completed, call this routine to disconnect from the remote host.

```
OSErr FTPViewFile(unsigned long connID,Ptr *file,char *fileName);
```

FTPViewFile retrieves a file from the remote machine and stores its data in a pointer allocated within the function.

```
OSErr FTPPutFile(unsigned long connID,char *fileName,char *data,
                 long size);
```

FTPPutFile is used to send a file to a remote machine. The remote filename and the file data and length must be given.

These routines are contained in FTPLow.c, using the same modular layered approach as for other protocols.

NEWSWATCHER CODE MODULARITY

As in the finger example, the NewsWatcher source code is structured for maximum flexibility in case of a protocol switch and to allow for ease in code sharing. The logical code blocks are shown in Figure 5. The FTPLow.c module, for example, could easily be extended and used as a generic file transfer module in many applications. It simply requires the medium- and high-level TCP calls described earlier.

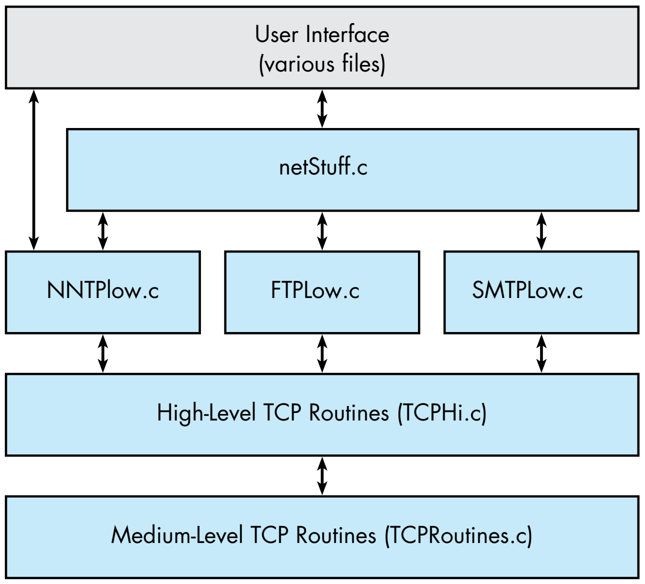

User Interface
(various files)
netStuff.c
NNTPlow.c
FTPLow.c
SMTPLow.c
High-Level TCP Routines (TCPHi.c)
Medium-Level TCP Routines (TCPRoutines.c)

Figure 5
NewsWatcher Code Modularization

SUMMARY

Although programming with MacTCP can be fairly complicated from a low-level point of view, using high-level routines makes it much simpler to create an easy-to-use TCP networking module. Once difficult issues, such as asynchronous behavior, are handled at a low level, higher-level modules can successively add protocol support for mail, news, and file transfer. These high-level routines can be used by any programmer, regardless of network-level knowledge.

For interested programmers, full source code to both the finger tool and NewsWatcher are available on the *Developer Essentials* disc.•

For information on the TCP/IP protocol stack and networking in general:

- Douglas E. Comer: *Internetworking with TCP/IP*, Prentice Hall, 1991.
- Andrew S. Tanenbaum: *Computer Networks*, Prentice Hall, 1988.

For an in-depth discussion of some of the protocols:

- J. B. Postel: Simple Mail Transfer Protocol, RFC #821, August 1982.
- J. B. Postel and J. K. Reynolds: File Transfer Protocol, RFC #959, October 1985.
- B. Kantor and P. Lapsley: Network News Transfer Protocol, RFC #977, February 1986.
- D. P. Zimmerman: Finger User Information Protocol, RFC #1194, November 1990.

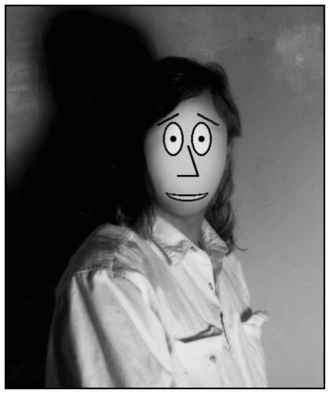
These RFCs are on the *Developer Essentials* disc. The library of RFC (Request For Comment) memos is available by anonymous FTP from NIS.NSF.NET in the directory RFC. Use FTP to connect to this host and use *anonymous* for username and *guest* for password.

A useful on-line reference for TCP/IP:

- Computer Science Facilities Group: *Introduction to the Internet Protocols*, Rutgers University, 1988.

This on-line reference is on the *Developer Essentials* disc.

Thanks to Our Technical Reviewers
Pete (Luke) Alexander, Brian Bechtel, Harry Chesley, Larry Rosenstein, Andy Shebanow, Gordon Sheridan, John Veizades.•



DAVE JOHNSON

THE VETERAN NEOPHYTE

A FAMILIAR (INTER)FACE

I just learned about a technique for graphically representing points in n-dimensional space, first presented in 1973 (I guess I'm a little behind) by a Harvard statistician, H. Chernoff. Representing points in two or three dimensions is pretty straightforward, but what about data points in, say, ten dimensions? Chernoff's approach was to use cartoon faces, with each dimensional parameter determining a facial feature. One parameter determines eye size, for instance; another regulates eyebrow slant, another determines the position of the mouth, and so on. I was struck dumb by the power of the idea. Humans have a built-in feature integration and recognition ability that lets us intuitively track and correlate changing facial features with no conscious effort, and Chernoff's technique elegantly capitalizes on this ability.

The possible uses for these "Chernoff faces" are many and varied. In his book *Computers, Pattern, Chaos, and Beauty*, Clifford Pickover talks about a number of possible applications: the elucidation of high-level statistical concepts, uses in air traffic control and aircraft piloting, educational applications, and many more. In one particularly illuminating example he uses the faces to characterize sound. He runs the sound data through an auto-correlation function (for the gory details, see his book) and then uses the first ten points of the resulting data set to control the faces. As examples he shows the faces generated by the sounds *s*, *sh*, *z*, and *v*, sounds which are very similar. The resulting faces, though, are easily and immediately distinguishable. As Pickover points out, this correlation between sounds and appearance of the faces immediately suggests the possibility of using the faces as feedback devices for helping severely hearing-impaired people to modify their vocalizations. To learn a particular sound, they would try to produce the same Chernoff face as that produced by a hearing person vocalizing the sound.

There are a few things worth noting about these faces. First of all, children respond just as well as adults to the faces. Since facial recognition capabilities develop in infancy, that's really not surprising, but it does mean that the faces have very broad application possibilities. I'd also bet that they're freer of cultural bias than many interface elements, which broadens their possible uses even more. Second, the faces probably aren't appropriate for quantitative analysis; if you need to get the exact value of a parameter, use a gauge. But they're great for high-level cognitive discrimination, especially for tracking qualitative changes in multiple variables through time. Third, because they're human faces, they could contain emotional connotations that have nothing whatever to do with the data they represent. Imagine a face used to track water levels and pump pressures in a nuclear power plant getting happier and happier as the plant approaches meltdown. Time for some facial calibration, I'd say.

Included on the *Developer Essentials* CD is my version of Pickover's face routine. It takes as arguments a pointer to a rect and a pointer to an array of 10 bytes, and it draws the resulting face in the current port, scaled to the rect. See the code for more details. One really cool thing is that if ten dimensions aren't enough, you can simply add more detail to the picture: hair maybe, or ears. The hardest part is coming up with meaningful 10-D data.

DAVE JOHNSON, our technical buckstopper, has been with Apple for three years. Before becoming an official stopper-of-the-buck, he worked on PostScript printers in Apple's software testing group. His interest in computers dates back to his college days at Humboldt State University, where he majored in energy systems engineering, and minored in everything else he could think of (it was a seven-year stay). He's also our resident juggler, who will juggle anything or anybody—and has. Actually, Dave has always been a show-biz kind of guy; as a kid, he was into monster makeup, which led him to monster-making and puppeteering at Lucasfilm's Industrial Light and Magic. (You've seen his work in such flicks as *Spaceballs*, *The Witches of Eastwick*, and *Inner Space*; he was also an On-Set Duck Mechanic for *Howard the Duck*.) When not buckstopping and juggling, he programs the Macintosh (screen savers, gratuitous fractal programs, artificial life simulations: nothing useful), hangs out with his wife and dogs, reads as much as possible, and pushes all available limits, both real and imagined.

What's really interesting is how this technique and others like it capitalize on the kind of processing that we already do automatically: you don't have to learn to discriminate faces; you already know how, and you do it without even trying. So even while some part of your brain is busily integrating facial features, your conscious mind is still free to deal with other, higher-level tasks. All the best interfaces do this to some degree, by simulating some part of what we call reality. (For instance, everyone's favorite desktop model simulates a flat, bounded environment with overlapping two-dimensional areas, something we're very familiar with in the real world through our interactions with tabletops and paper.) A convincing simulation, or even one that captures essential parts of the reality (like the Macintosh desktop), is an incredibly powerful thing.

Human interfaces (so far humans are the predominant, market-driving users of computers) can take advantage of many things: our ability to maintain internal mental maps, our built-in image processing, our kinesthetic awareness of space, and so on. All these things have been finely tuned by a zillion years of evolution, so why not use them? Effective human interfaces are overwhelmingly visual and tactile, precisely because vision and touch are the primary senses we use to interact with the world we know so well. (For a dog I suppose a good interface would have to be heavily auditory and olfactory: sniff a file to get info; when something you search for is found, it whines so you can locate it; system errors smell like flea shampoo.)

New interfaces are trying to capitalize even further on what we already know how to do. Xerox PARC's latest experimental interface, the Information Visualizer, uses 3-D real-time animation and represents information as directly manipulable 3-D objects. You can "pick up" a data structure and look at it from all angles, using your built-in spatial skills to help make sense of large bodies of information. PenPoint, Go Corporation's recently announced pen-based operating system, takes advantage of, among other things, our familiarity with pencils and notebooks. (I guess it's not recent anymore, is it? It's still February in here.)

The goal of an interface is to make using the computer easier and more intuitive. What more direct way than to simulate on the computer things that people already know how to work with? A really interesting question is whether simulation is a necessary part of a good interface, but it's a question without an answer yet, at least as far as I know. In the meantime, while you're programming your next whiz-bang interface, remember to occasionally look beyond the next crash or whether you remembered to unlock that handle. There are many, many things that all people are already good at: take advantage of them.

RELATED AND SEMI-RELATED READING

- *Computers, Pattern, Chaos, and Beauty* by Clifford A. Pickover (St. Martin's Press, 1990).
- *A Monster Is Bigger Than 9* by Claire and Mary Ericksen (The Green Tiger Press, 1988).
- *Byte*, February 1991.
- Anything by Bruce Tognazzini.

MACINTOSH Q & A

Q *The TrapAvailable function listed in the "Compatibility Guidelines" chapter of Inside Macintosh Volume VI contains code that checks the size of the trap table. Do I have to do this if my application doesn't support the old 64K ROMs?*

A Yes, you do. With the introduction of the Macintosh Plus and 128K ROMs, the trap table was divided into the OS trap table and Toolbox trap table. The number of trap table entries was increased and the format of those entries was expanded. For additional information on this expansion see the "Using Assembly Language" chapter of *Inside Macintosh* Volume IV. However, the 64K to 128K ROM trap table expansion is not what the code in the TrapAvailable function is checking for. The Toolbox trap table was expanded yet again when the Macintosh II was introduced. The Macintosh Plus and SE use the smaller Toolbox trap table on pre-7.0 systems. To be sure your application is compatible with these machines on pre-7.0 systems, check the size of the Toolbox trap table.

Q *What are the guidelines for determining how much of an image CopyBits can copy to a Macintosh pixel map at one time, given a particular set of characteristics for the source map and the destination map and given how much stack space is available? For example, say that we have an 8-bit-deep pixMap to be copied to a 32-bit-deep pixMap using the ditherCopy mode and expanded by a factor of 4, and we have 45K of stack space.*

A CopyBits' stack requirement depends on the width of each scan line (rowBytes). The rule of thumb is that you need at least as much stack as the rowBytes value in your image (which can be huge with 32-Bit QuickDraw), with the following additional modifiers: add an additional rowBytes for dithering; add an additional rowBytes for any stretching (source rect != dest rect); add an additional rowBytes for any color map changing; add an additional rowBytes for any color aliasing. The stack space you need is roughly five times the rowBytes of your image. In general, you're better off processing narrower scan lines. Reducing the vertical size will not affect stack requirements. Narrow, tall bands (if you can use them) will reduce the stack requirements.

Q *Where can I find documentation on how to write a Macintosh printer driver equivalent to the ImageWriter® or LaserWriter driver? In particular, how are Printing Manager and QuickDraw commands translated into calls to the printer driver?*

A DTS's "Learning to Drive" document and "SampleWriter" source code, available in AppleLink's Developer Support folder and on the latest *Developer Essentials* CD, are helpful references.

Kudos to our readers who care enough to ask us terrific and well thought-out questions. The answers are supplied by our teams of technical gurus; our thanks to all. Special thanks to Pete "Luke" Alexander, Mark Baumwell, Rich Collyer, Marcie "MG" Griffin, C. K. Haun, Dennis Hescox, Kevin Mellander, Guillermo Ortiz, Keith Rollin, Kent Sandvik, Gordon Sheridan, Paul Snively, Bryan Stearns, Forrest Tanaka, Vince Tapia, and Scott "Zz" Zimmerman for the material in this Q & A column. •

Q *How can I determine what hardware device is driven by a particular Macintosh gDevice? I can call GetDeviceList and GetNextDevice to get the driver reference number of each gDevice but not the hardware ID. The system 'scrn' resource for the hardware ID of each device doesn't give me the driver reference number. The device list produced by calling GetDeviceList and GetNextDevice isn't always in the same order as the 'scrn' resource slot information.*

A The following code shows how to obtain the slot number:

```
DCEHand = (AuxDCEHandle) GetDCtlEntry(DevInfoList[index].gdRefNum);
DevInfoList[devCount].gdSlot = (*DCEHand) -> dCtlSlot;
                                             /* Get slot number. */
```

Once you have the slot number, you can call the Slot Manager to get the board name and other information you may need to identify the device, as in the following code:

```
/* Get the board name from the Slot Manager's board sResource. */
spB.spSlot = DevInfoList[devCount].gdSlot;
spB.spID = 0;
spB.spExtDev = 0;
spB.spCategory = 1;             /* board sResource sRsrcType value */
spB.spCType = 0;
spB.spDrvrSW = 0;
spB.spDrvrHW = 0;
if (! SNextTypesRsrc(&spB) ) {
  spB.spID = 2;        /* Found board sRsrc now, get the sRsrcName. */
  if ( ! SGetCString(&spB) )    { /* Let C unravel its own strings. */
      for (count = 0;
           DevInfoList[devCount].bdName[count+1]
                 = *((char *)(spB.spResult)+count);
           count++);
       DevInfoList[devCount].bdName[0] = count;
  }
}
```

Q *When I draw a 32-bit Macintosh 'PICT' image from a file to an 8-bit port via an off-screen GWorld, I use dithered mode in the CopyBits call and the results are quite impressive. If there's not enough memory to allocate the GWorld, I draw the image directly to the port. But since there doesn't seem to be any way to tell QuickDraw to use dithered drawing mode, the image looks horrible. Should I use dithered mode instead of source mode? I don't want to try to parse the 'PICT' myself, but I thought that maybe a QuickDraw global could be modified in my StdBits proc to force dithered drawing for that operation only.*

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

A You can install a bitsProc bottleneck procedure to get all the CopyBits calls when the picture is being played back. One of the parameters to the bitsProc call is the mode. You can install a procedure that passes ditherMode to the original StdBits proc.

Q *Why does a call to gestaltNuBusConnectors return zero slots when there actually is a NuBus™ card or a PDS (Processor Direct Slot) card in the Macintosh IIsi slot?*

A A call to gestaltNuBusConnectors returns zero slots because there's no way to determine if there's a NuBus card or a PDS card in the Macintosh IIsi slot. Gestalt can't assume that there's always a NuBus or PDS slot, so it just says there's no slot. However, Apple recommends that you always use the Slot Manager instead of Gestalt to search for cards, after first checking to see that the Slot Manager trap is implemented. The Slot Manager will safely do all the necessary work for you whether a card with a valid declaration ROM is installed or not, and you can search for the card using a variety of criteria. This technique will allow you to locate NuBus cards, but has the added benefit of being able to find PDS cards that contain declaration ROMs.

The Macintosh SE/30, IIfx, and LC have the Slot Manager and PDS slots. It's safe to assume the Slot Manager will be resident in all future machines that have either NuBus or PDS slot capability.

Q *Do you have any available tools or test programs for testing how well our Macintosh application responds to core Apple events? We need a test application that will, for example, send events to open a document and print it within our application.*

A One of the best tools to test Apple events like 'odoc' and 'pdoc' is the Finder®. The System 7.0 beta CD version of the Finder provides excellent 'odoc' and 'pdoc' testing capabilities. That's what most of DTS uses to test sample applications.

There are no tools designed specifically to send Apple events to an application as a test. Try using some of the DTS sample applications on the *Developer Essentials* disc to test your application. One of the easiest to use and adapt is the TrafficLight 3.0 sample. It already will send its own MoveWindow Apple event to any application, allowing you to see how your application is handling "foreign" events.

Also, you can adapt the CreateAndSendAppleEvent function in AppleEvents.c to send any type of event you'd like, which allows you to test events that may be specific to your application.

Q *Why do I get a bomb when I create a Macintosh filename starting with a period (.)?*

A Macintosh filenames are not allowed to begin with a period, to avoid possible confusion with driver names, which must begin with a period. (This restriction does not apply to folder names.) Ideally, the Finder should catch this possible error and require the file to be renamed, but it doesn't. Future versions of the Finder should catch this potential problem, but until then users must remember not to begin a filename with a period. See Macintosh Technical Note #102, HFS Elucidations, for details.

Q *Can the refNum returned by FSOpen ever be 1? What is the range or format of legal refNums?*

A Macintosh file reference numbers (refNums) are currently positive and that means any positive number, including 1. This doesn't mean that they won't change in the future, however. To maintain system compatibility, use refNums only as they're intended to be used.

Q *Calling Create from an INIT causes the INIT to be called twice if it alphabetically follows the file it creates. Is there a workaround?*

A This particular limitation of the startup process occurs when you create files in the same folder as the INIT that alphabetically precede your INIT file. HFS orders files alphabetically, and the startup process does its thing by incrementing the ioFDirIndex field of a PBGetFileInfo call. Your INIT file (for example, file #5) is getting opened, your INIT is called, and it creates a file. The file it creates, for example, is file #3—that is, it's alphabetically before your INIT file. Now your INIT file is file #6, and when you return, it increments the ioFDirIndex value from 5 to—you guessed it—6, opens that file, and runs your INIT again.

The workaround is to avoid creating files in the same folder as the INIT that alphabetically precede your INIT file. Also, the file you create should be placed in the Preferences folder within the System Folder, for both System 6 and System 7. Check for a Preferences folder within the System Folder using FindFolder, and create one if it doesn't exist.

Q *During a Macintosh application's life, does the value of A5 change? Why does SetCurrentA5 have to set A5 to CurrentA5? Aren't A5 and CurrentA5 the same while an application is executing non–interrupt-time code?*

A A5 is not necessarily always equal to CurrentA5. Because the Macintosh operating system and Toolbox don't need to access your application's jump table or global variables (which A5 points to), they often use A5 for other purposes, except for the parts of the Toolbox that need to perform graphics operations and use the QuickDraw globals defined by your application.

Because the operating system or Toolbox can change A5, you must make sure it's set correctly if the operating system or Toolbox ever calls your code in the form of a callback. For example, if you make an asynchronous File Manager call, your I/O completion routine must call SetCurrentA5 for your I/O completion routine. Other places where you may want to call SetCurrentA5 are in trap patches, your GrowZoneProc, custom MDEFs, WDEFs, or CDEFs, or control action procedures.

Please note that even SetCurrentA5 is not sufficient when writing code that's executed at interrupt time, such as VBL and Time Manager tasks. When an interrupt occurs that your application wants to handle, there is no guarantee that the low-memory global CurrentA5 belongs to your application. The interrupt could occur while some other application is running under MultiFinder®. In this case, you should use other approaches to setting A5. Please see Technical Note #180, MultiFinder Miscellanea, for further information.

Q *My Macintosh application receives information from the serial port, keeping as much information in memory as possible. However, if I do run out of memory, my GrowZone routine is allowed to delete older information. Does the operating system allow the GrowZone routine to be called recursively? That is, if in handling a GrowZone call, my code does something that requires memory, such as bringing in a font that isn't there, can I call the GrowZone routine again or would doing so cause an error? Am I allowed to update things on the screen, such as windows and text in windows, while in my GrowZone routine?*

A Neither the GrowZone routine nor the Macintosh system is reentrant. There is very little that you can safely do in a GrowZone routine. The system expects that the typical extent of a GrowZone routine is to check a table or some other predesignated criteria and then to release or not to release any memory blocks it can. Any sort of user interface is well beyond the reasonable scope of this level of routine.

Keep a list of handles to blocks of text that you're willing to purge, so that your GrowZone routine can look through them when called and then release one or more blocks. Any more than that is really stretching GrowZone too far.

By the way, SetGrowZone should not be assumed to return a result. *Inside Macintosh* is incorrect in saying that D0 contains a result code on exit.

Q *Do I need to call StripAddress when using SwapMMUMode to switch to 32-bit mode? I would like to assume that all my pointers are valid 32-bit quantities when in 32-bit mode.*

A Yes, though it would be nice to be able to assume that all your master pointers are valid 32-bit quantities when in 32-bit mode, the assumption would only be true of master pointers that were created in a zone created while you were in 32-bit mode, and even this would only be true on systems that have a 32-bit Memory Manager (the Macintosh IIci, IIfx, LC, and IIsi).

You need to call StripAddress on any master pointers that were created in a 24-bit world that you expect to use while in 32-bit mode. You also need to call StripAddress on distinct master pointers that were created in 24-bit mode if you want to compare them, regardless of whether you're currently in 32-bit mode.

Q *When building a standalone code resource (such as an 'XCMD', 'INIT', 'CDEV', or 'CDEF') with the MPW Linker, the main entry point is specified with the "-m" option. So why does one also need to make sure that the main routine is the first one in its object file, and that that object file is the first one linked? Why isn't "-m" enough for the Linker to figure this out?*

A The MPW® Linker generally places modules in the output file in the order that they appear in the sequence of files to be linked. When building an application, the Linker builds the jump table after building all the code resources; thus, it doesn't need to have "seen" the main procedure first in order to place the main procedure's entry first in the jump table. For standalone code resources (which have no jump table), the main procedure must be first in the code resource. The simplest way to make sure this happens is, as you say, to make sure that the main procedure is the first one the Linker sees.

Q *Why does MPW Pascal convert SINGLE numbers to extended, create a temporary area for them, and then pass a pointer to those extended numbers on the stack?*

A The conversion of single-precision values to extended is being done to maintain accuracy. It's entirely possible to generate values of extended precision while doing the intermediate math with single-precision values, and the MPW compilers do this conversion to preserve accuracy.

If your application doesn't need the accuracy, you can declare the parameters to be of type LONGINT, and typecast them as necessary within your procedure or function. There's no way to tell the MPW compilers not to do the conversion if the parameters are declared as SINGLE.

Q *What is the difference between RelString and EqualString? What should Macintosh developers use when sorting? Do you suggest having an option for the international sort?*

A RelString and EqualString are mainly intended for the File Manager. The File Manager uses them for quick-and-dirty string comparison so that it knows how to return files ordered alphabetically when you use indexed File Manager routines, and so that it can detect file name collisions. Beyond that, RelString and EqualString aren't localizable or extensible.

The International Utilities string comparison routines are localizable and extensible. They use information in the active 'itl2' resource to determine how the characters are sorted. Most localized systems come with their own 'itl2' resource, so string comparisons are done correctly for the region for which the system is localized. Because RelString and EqualString stay the same for all these regions, you'll probably find some cases in which strings are compared incorrectly by these routines.

One important place where RelString and EqualString don't work very well is with the new characters in the extended Macintosh character set. When the LaserWriter was introduced, the LaserWriter fonts used the extended Macintosh character set, which added many new characters, including several new uppercase characters with diacriticals. In system software version 6.0.4, the International Utilities were updated to take advantage of these new characters. For example, the uppercase "E" with a grave accent first appeared in the extended Macintosh character set. With 6.0.4, the lowercase and uppercase "E" with a grave accent were considered to be equal in primary ordering, and unequal in secondary ordering, which is correct. Even today, RelString and EqualString think in the old Macintosh character set, erroneously believing that lowercase and uppercase "E" with a grave accent have nothing to do with each other.

Q *The Macintosh IIsi and LC computers don't come with programmer's switches. How do I get reset and NMI on these machines?*

A Both machines have an ADB (Apple Desktop Bus) I/O processor which incorporates the reset and NMI functions through the keyboard. The functions are accessed as follows:

- Use MacsBug 6.2 or later to enable the NMI on the Macintosh IIsi or LC. Both machines start up with NMI disabled because nontechnical users might get into trouble with it, so the ADB controller must be programmed to enable it. Once MacsBug is installed, you can activate NMI by holding down the Command key while pressing the Power button.

- Hold down the Command and Control keys while pressing the Power button, with or without MacsBug installed, to reset these Macintosh systems.

Under some circumstances you may have to hold the reset and NMI key combinations down for a little while (but no more than a second) to make sure the ADB processor sees them.

You'll find the latest MacsBug on the *Developer Essentials* disc and in AppleLink's Developer Support folder. The MacsBug reference manual is very helpful. A package containing both MacsBug 6.2 and the manual is available from APDA for $35.00 (#M7034/B).

Q *What's wrong with having VBL (Vertical Blanking) tasks make calls to the Macintosh Memory Manager, either directly or indirectly?*

A The problem is that the Memory Manager could be moving memory around when an interrupt occurs. If the VBL task also moves memory, the heap could be destroyed.

VBL tasks are intended to provide a method of time-syncing to the video beam of the display. (On slotted Macintosh models you'd use SlotVInstall.) They're also used to get periodic time for short tasks, although the Time Manager is better for this. VBL tasks should minimize execution time. The best use of a VBL task is to do a short condition check and set a flag for the main process to indicate that it's now a good time to do something.

Q *Can I still write to James Brown at his work release address?*

A The Godfather of Soul was released from prison on parole last March. His new address is

Universal Attractions
218 West 57 Street
New York, NY 10019

APPLE II Q & A

Q *When I write an Apple IIGS TextEdit keyFilter procedure and put its address in my TextEdit control template, I get funny little pieces of garbage drawn on the screen in my TextEdit record, and sometimes TextEdit crashes. Doesn't the keyFilter mechanism work?*

A The keyFilter mechanism in TextEdit works, but there's no space for a keyFilter address in a TextEdit control template. The only filter procedure in the control template is the *generic* filter procedure, which does not take the same parameters as the keyFilter procedure. If you include a filterProc address in a TextEdit control template, it must be to a generic filter procedure. Generic filter procedures are defined on pages 49-16 through 49-18 of *Apple IIGS Toolbox Reference* Volume 3.

If you want to use TextEdit's keystroke filter, word wrap hook, or word break hook, you must modify the TERecord directly to put your procedure's address in the appropriate place.

Q *While linking my Apple IIGS® application, LinkIIGS does a system death: "ExpressLoad error 1301." What am I doing wrong?*

A Believe it or not, LinkIIGS has dynamic segments. ExpressLoad has the annoying habit of taking error codes it's not expecting, adding $1100 to them, and calling SysFailMgr. Why is this habit annoying? Well, it works just fine for what the author had in mind, which was GS/OS errors (all of the form $00xx), but it causes problems with Toolbox errors.

You've probably figured out the rest by now: $1301 - $1100 = $0201 = Memory Manager "unable to allocate handle" error. ExpressLoad ran out of memory trying to load a dynamic segment in LinkIIGS.

The best solution is to get more memory. If you can't make enough memory to link it, you might consider having some of your code in code resources instead of in dynamic segments in the data fork, to create separate links for those segments and make the big link even smaller.

Q *What is the correct procedure for installing system software on an Apple IIGS?*

A Always use the Installer application. The Installer scripts provided by Apple on the Apple IIGS system disks will put everything you need on the startup disk. If you attempt to install system software without using the Installer, there's a good chance you'll forget to copy a needed file or delete an obsolete file.

Kudos to our readers who care enough to ask us terrific and well thought-out questions. The answers are supplied by our teams of technical gurus; our thanks to all. Special thanks to Matt Deatherage, C. K. Haun, Jim Luther, and Jim Mensch for the material in this Q & A column. •

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

Because SCSI hard disk drivers are not included on System.Disk, you'll need to do the following to install System 5.0.4 on a hard disk:

1. Make backup copies of System.Disk and System.Tools, making sure they keep the same names.

2. Launch the Installer from System.Tools (backup) and install SCSI Driver on the backup of System.Disk. The Installer script will delete a couple of fonts as well as the tutorial folder to make room for the SCSI information on a 3.5-inch disk.

3. Boot the backup of System.Disk and install System 5.0.4 onto your hard drive. If you want Shaston 16 and Times 12 fonts, install "Additional Fonts" last.

The Installer knows how to make the software fit on a floppy disk. It can also update your system without requiring you to trash your existing System Folder.

Apple IIGS Technical Note #64, Apple IIGS Installer and Installer Scripts, describes how the Apple IIGS Installer executes Installer script files and how to write Installer script files.

Q *In some of my recent work, I've found it necessary to patch the Apple IIGS GS/OS vectors in order to monitor OS calls. My patch works without interfering, but it disappears when the user switches to ProDOS 8 and back to GS/OS. I tried unsuccessfully to fix this by using the notification queue, asking GS/OS to notify me when the user was coming back from ProDOS 8. How can I safely and reliably patch GS/OS on a permanent basis?*

A You should be able to patch GS/OS® with System 5.0.4. Two observations may help:

- If you ask for all notification events, you'll get at least one disk-insert event before you get any restart events, since the device driver for the startup disk will "fake" a disk insert to get the appropriate disk-switched statuses set before doing any real work.

- Your procedure may not be the first notification procedure called at restart time. For example, the Resource Manager inserts a procedure so that it can reopen the system resource file on return from ProDOS® 8. If this mechanism was broken as you say, the Resource Manager's notification procedure wouldn't work either.

Q *Why does a dialog box without a Cancel button come up from an Apple IIGS Loader call when the volume is not on-line?*

A The loaders always set the preferences to “dialog, no Cancel button” when trying to load a dynamic segment indirectly (because you passed control to it). The loaders must do this because they have no way to report errors. For example, if your code does a JMP DynSegLabel, the Loader must load the dynamic segment. Should it get an error, it has no way to report an error and no place to pass control back to if your program does a JMP and the Loader has no place to return to. In earlier systems, inability to load a dynamic segment was a fatal system error. Today, the Loader will not give up until you insert the disk because it has no other choice. However, if you call LoadSegName yourself, the Loader should not change the preferences; because that’s a call, the Loader can return from it gracefully. Indirect dynamic segment loading doesn’t have that luxury. The current Loader documentation is in the Addison-Wesley version of *GS/OS Reference*.

Q *What are longStatText2 items, mentioned in the “Dialog Manager” chapter of the* Apple IIGS Toolbox Reference?

A A longStatText2 item is similar to a longStatText item except that the text is drawn with LETextBox2. A longStatText2 item allows you to embed formatting codes so that you can change fonts, font styles, sizes, colors, and justification. The longStatText2 capability is built into the Dialog Manager to support formatting flexibility in standard dialogs. To use longStatText2 items, the Apple IIGS QuickDraw Auxiliary and Font Manager tools must be started.

Q *Where do I find a current list of the MessageCenter message types that have been registered with DTS?*

A Message types that are assigned to individual developers are treated confidentially. We have very few of these, as most developers now use MessageByName to get an assignment dynamically.

Q *Where do I find technical documentation on the messages written by the Finder to tell an Apple IIGS application which files to open as it starts up?*

A The only truly public message types are #1 (the files message from the Finder or other program launchers) and #2 (the desktop pattern message).

Message type #1 is documented in *Apple IIGS Toolbox Reference* Volume 2, with the description of the MessageCenter tool call. The documentation implies that there are both filenames and full pathnames (such as Standard File returns) in the message, but, in fact, each Pascal string indicates a totally separate file. Message type #2 is documented in *Apple IIGS Toolbox Reference* Volume 3, in the "Window Manager Update" chapter (page 52-4).

Q *How does an Apple IIGS New Desk Accessory (NDA) obtain its ID?*

A Each Apple IIGS NDA has two IDs: a Memory Manager ID and a Menu ID. If you want the Memory Manager ID, simply call MMStartup from your DA. The NDA ID for the OpenNDA call can easily be obtained from your menu string. The Desk Manager replaces the ** of your \H** at the end of your menu string with your Menu ID, which is also your NDA number. A note of caution: Please be sure you're running in the 16-bit environment before using the NDA ID to call OpenNDA. If you try this while ProDOS 8 is running, nothing good comes of it!

Q *I'm creating an Apple IIGS list control from a resource. How can I update my listRef resource dynamically, if my list grows dynamically?*

A To modify the content of a resource and to grow it, load the resource, make any changes you want to the handle (such as change the data inside or call SetHandleSize to make it bigger), and then use the MarkResourceChange call to tell the Resource Manager your resource's content has been changed. The Resource Manager then updates the contents of your file when you call UpdateResourceFile. The Resource Manager even recognizes handle size and content changes (actually, it just assumes the contents have changed).

Q *Why is there no GS-only version of HyperMover?*

A HyperMover™ is actually implemented as a pair of HyperCard® stacks: HyperMover.mac, which runs under Macintosh HyperCard and disassembles Macintosh stacks; and HyperMover.GS, which runs under HyperCard IIGS and reassembles the stacks into HyperCard IIGS stacks. Although there are a few XCMDs to handle tricky stuff like sounds and paint files, the majority of the work is done by simple HyperTalk scripts. It may seem like a disadvantage to require two computers to do the translation, but in fact there are a number of tremendous advantages:

- HyperMover doesn't need to know anything about the internal binary format of stacks. This makes it somewhat immune to stack format changes. For example, you can convert a Macintosh HyperCard 1.2.5 stack to HyperCard 2.0 format, but HyperMover will still translate it because the HyperCard program takes care of reading data from the stack.

- The interchange format is simple—a file containing a complete textual description of the stack. After disassembly, you can open and even edit this file using any text editor (such as MPW or APW), before reassembling the file on the GS side. This provides an easy way to browse scripts, looking for potential machine dependencies, and you can actually perform global modifications on your stack using the find-and-replace capabilities of your text editor.

- When small-font painted text is used to label objects, it often shrinks to unreadability when converted to the Apple IIGS screen resolution. You'll need a Macintosh to view all the graphics and decide what they're supposed to look like, before you can redraw them in 16-color Apple IIGS graphics.

- You'll also need a Macintosh to perform side-by-side comparisons and testing of your new stack.

Having a Macintosh available is important at *all* phases of the stack translation process. You'll find it makes the entire process much smoother.

Q *Which versions of Macintosh HyperCard are compatible with HyperMover?*

A The HyperMover stack will execute in the 1.2.5 and 2.0 Macintosh HyperCard environments. However, stacks that are translated should be either 1.2.5 stacks, or 1.2.5 stacks *converted* to 2.0 but not *modified*. There are two reasons for this:

- The graphics converters are designed to start from a Macintosh HyperCard 1.2.5–sized card only, because stacks with other card sizes may have objects and graphics improperly aligned.

- HyperCard IIGS uses a version of the HyperTalk® scripting language derived from Macintosh HyperCard 1.2.5, so if a stack uses language elements that were not present in Macintosh HyperCard 1.2.5, it may translate correctly but report script errors when the script is executed.

Q *How can I perform error trapping in a script?*

A Normally, most errors interrupt script execution and immediately present a dialog to the user. Advanced scripters may want to intercept these errors and deal with them more "aggressively." Two new HyperCard properties, lockErrors and lastError, have been provided to control error handling:

- lockErrors is a Boolean property that has a simple effect: it prevents the display of errors. The errors, however, are still there, and stop execution of the current handler.

- lastError is a string property that always contains the text of the most recent error dialog, whether displayed or not. Because errors cause handlers to terminate, you'll probably wind up checking lastError from an idle handler.

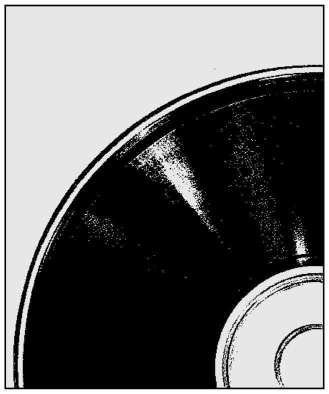
How do you use lockErrors and lastError? To activate an "error catcher" you can use the following scripts:

```
on CatchErrors errorSource    -- Begin handling errors.
  global gErrorSource
  set the lastError to empty
  set lockErrors to true
  put errorSource into gErrorSource
end CatchErrors
on ClearErrors        -- Handle errors normally.
  global gErrorSource
  set the lastError to empty
  set lockErrors to false
  put empty into gErrorSource
end ClearErrors
on idle               -- Check for error occurrence.
  global gErrorSource
  if gErrorSource is not empty
  then
    -- Error-handling code goes here.
    -- gErrorSource = where in your code the error happened.
    -- The lastError = what the error was.
  end if
  pass idle
end idle
```

If you have a button that does something dangerous, you can surround the dangerous portions of the handler with

```
on mouseUp
  -- normal error handling out here
  CatchErrors "dangerous operation #1"
  -- Do things that might not work.
  ClearErrors
  -- Now we're back to normal error handling.
end mouseUp
```

For more information, be sure to see the *HyperCard IIGS Script Language Guide*, published by Addison-Wesley.



YOUR DEVELOPER ESSENTIALS DISC

WHAT'S OLD AND WHAT'S NEW

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

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

We start out by describing the old standbys (and what they're standing by for) and finish up with descriptions of what's new or improved on this disc. We're especially happy to introduce snippets in this issue—read on to find out why.

THE STANDBYS

develop

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

SpInside Macintosh

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

DTS Technical Notes and Sample Code

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

Macintosh Technical Notes Stack

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

Q & A Stack

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

Human Interface Notes and Stack

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

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

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

U.S. System Software/HyperCard
Here you'll find system software versions from 0.1 to 6.0.5—you can copy them right to a floppy disk using DiskCopy. You'll also find HyperCard U.S. versions 1.2.2, 1.2.5, and 2.0, all of which come complete with an idea stack. Have you ever wondered how many gills there are in a pint? Find the answer in the idea stack.

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

NEW OR IMPROVED

Q & A Stack
You've told us in surveys and in focus groups that the Q & A Stack just doesn't work for you: the format's ugly, the information's out of date, and there aren't enough question and answer pairs to make it worth your while to look there for answers. Well, we listened. The new, improved, and tremendously beefed up (or radished up, for you vegetarians out there) Q & A Stack has changed so much so that we toyed with changing the name to get you to look at it again. It's organized more clearly, the information's up to date, and there are hundreds of question and answer pairs available for your searching pleasure.

System 7.0 Sample Code
This is the most robust code from the System 7.0 Beta 4 CD; if you've taken the samples from there, you don't need these, since they're the same. System 7.0 was still in its Beta cycle when these samples were written. That means that none of these samples are in their final form and there may be bugs or some incomplete sections—look for later updates on AppleLink and on future versions of *Developer Essentials*.

CShell She sells C Shells by the MPW shore . . . no, not really. CShell is a complete Macintosh application shell that includes the basics that are common to all Macintosh applications (whatever system version you're targeting for). These basics include an event loop, menu handling, window handling, TextEdit, file handling, and basic printing code. This sample also includes Apple-event code (both required and custom) for System 7.0 applications. For Think C enthusiasts, we've included a Think C version of CShell.

DTS.Utilities This is a grab bag of useful routines that any self-respecting application would like to have. There are many useful window, dialog, QuickDraw, and control functions included in this package. Browse through the header files and look at the functions available; you'll probably see something you need or can use. CShell and Kibitz (described below) both require these routines, so you'll need to have them available if you build those projects.

Edition Manager Read All About It! The Edition Manager is not as fearsome as you may think! This sample shows you the basics of using the Edition Manager with PICT and TEXT data types in a multiwindowed application. It shows you how to use the Edition Manager routines, plus one method of dealing with the internal bookkeeping of section handles. Browse the code, cut out the pieces you can use, and make your application Edition Manager aware. This sample also shows you how to handle the four required Apple events, plus the Apple events that are specific to the Edition Manager.

INIT - CDEV This sample uses the PPC toolbox to communicate between an INIT and a cdev, something that has never had a very satisfactory solution until System 7.0. It also demonstrates some very good techniques for writing INITs and cdevs. If your package requires special INITs or cdevs, you can save yourself a lot of work by looking this over first.

Kibitz It's your move. Kibitz puts CShell to work by building a working Apple-event–reliant application on top of the shell. Kibitz is a two-player chess program that plays across your network, communicating through Apple events. The move-passing code will be of special interest to those of you who need to implement a private Apple event type. Caution: Please try to actually look at the code, and don't spend all your time playing the game!

ProcDoggie This sample lets you dog the heels of any running process, and shows how to work with the new Process Manager calls, as well as showing you how to use the new LaunchApplication trap. ProcDoggie shows you all the current processes on your machine, and lets you examine, launch, and kill any process. Besides showing how to use Process Manager calls, ProcDoggie can be very handy during debugging sessions.

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

Audio CD An MPW Tool that allows you to start, stop, pause, and continue audio CD tracks.

BusErrorTest Shows how to replace the 68K bus error vector—very useful for testing.

ChangeTextStyleRec A utility routine that can simplify the process of modifying a TextStyle record to change a font, style, or type size.

ClickSound A simple sound producer.

ClutWind Displays a window that shows the colors in the color table associated with the device the window is on top of.

DisableEject Shows how to stop a floppy disk from being ejected. We've provided this snippet because many of you have asked how to do it, but you should know as well as we do that it has no chance of being compatible with future system software.

Heap Purge dcmd Simulates TMON's heap purge in MacsBug.

Icon Display A Think C project that reads pixMaps from an icon family (icsX) that are actually stored as pure pixel data (not a PixMap struct or a color icon), allocates an appropriate off-screen world, and then copies the pixels into the off-screen pixMap with BlockMove. This creates a pixMap you can use with CopyBits. This code is found in the bullwindow.c file. There's some other code that demonstrates how to keep the pixMap matched up to the depth of the current device.

GetFInfo, GetVInfo MPW Tools that simply parse command-line options and print the value of the parameter blocks returned from Toolbox calls.

KeyMapTest Shows how to interpret the results of GetKeys.

Thanks to C. K. Haun for collecting and describing the System 7.0 samples and to C. K. Haun, Pete Helme, Jim Reekes, Keith Rollin, Eric Soldan, and Forrest Tanaka for writing them.•

Marquee Demonstrates marching ants (the scrolling dashed lines used in a selection rectangle).

MDEF.Sample An MDEF written in Pascal that supports rez MENU templates and allows you to request the use of the Shift-Command symbol (made famous by our friends to the North and perhaps MacroMaker) for a menu item instead of the traditional Command-key symbol. The MDEF also erases and redraws items rather than inverting them; this is necessary for a hierarchical menu.

OpenWindow Shows basic initialization calls and how to open a window for drawing.

Process An MPW tool that prints information about all running processes under System 7.0.

ReadLN.c A routine that causes the FSRead call to behave like a Pascal Readln.

ReKeyTrans Shows how to use the Script Manager and KCHRs to call KeyTrans.

TCP Includes two MPW Tools, TCPSend and TCPReceive, that demonstrate a very simple establishment of a TCP connection and sending a text string over that connection. An API library of all the MacCTB driver and DNR calls is also included.

TickAnimate Sample of how to use ticks to synchronize drawing to the screen.

TimerTst This hardware-dependent snippet shows how to use the VIA timers (for the few times when the Time Manager is not appropriate).

ZoomWindow Demonstrates how to properly zoom a window. It gives attractive results with most WDEFs because it uses the window's structure region rather than just its portRect.

There are also a number of esoteric NBP and PPC samples that demonstrate, among many other wonderful things, how to use the PPC toolbox.

Thanks to Guillermo Ortiz for collecting and making sense of these snippets, and to James Beninghaus, Cameron Birse, Tim Enwall, John Harvey, Dennis Hescox, and Craig Prouse for submitting them. •

INDEX

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