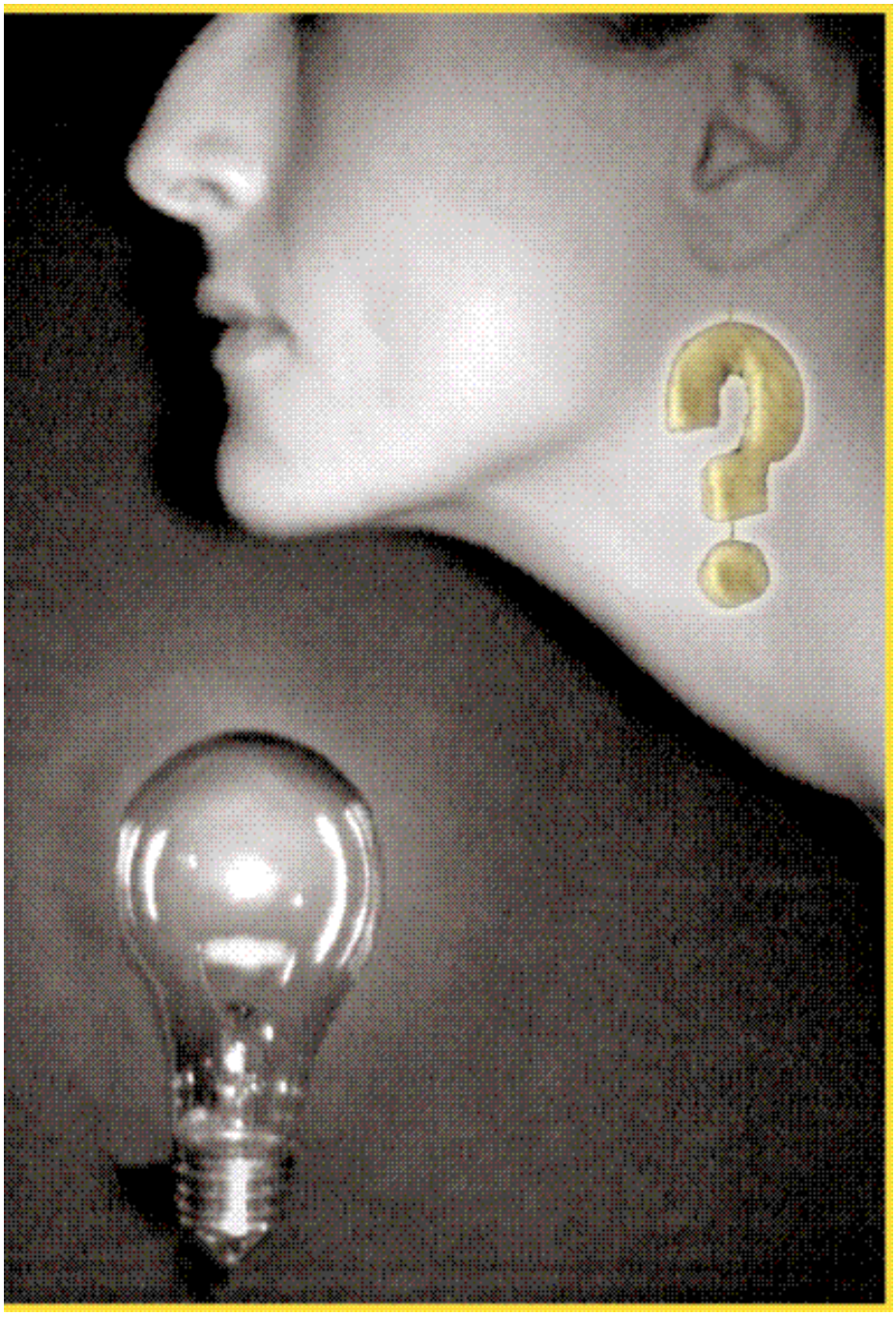
develop

The Apple Technical Journal



Vol.2, Issue 1 Winter 1991

Apple Computer, Inc.

EDITORIAL

Editor in Chief's Clothing *Louella Pizzuti*
Technical Buckstopper *Dave Johnson*
Managing Editor *Monica Meffert*
Developmental Editors *Lorraine Anderson, Judy Bligh, Judy Helfand, Loralee Windsor*
Editorial Assistant *Patti Kemp*
Copy Editor *Toni Haskell*
Production Manager *Hartley Lesser*
Indexer *Ira Kleinberg*
Manager, Developer Technical Communications *David Krathwohl*

ART & PRODUCTION

Design *Joss Parsey*
Technical Illustration *J. Goldstein*
Formatting *Bruce Potterton*
Printing *Craftsman Press*
Film Preparation *Aptos Post, Inc.*
Production *PrePress Assembly*
Photographer *Ralph Portillo*
Circulation Management *Dee Kiamy*
Online Production *Cassi Carpenter*

REVIEW BOARD

Pete "Luke" Alexander
Larry "Cat Couch" Rosenstein
Andy "The Shebanator" Shebanow

To create the cover, Hal Rucker, Cleo Huggins, a flashlight, black construction paper, a lightbulb, a chair and a whole lot of duct tape came together.

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

CONTENTS

© 1991 Apple Computer, Inc. All rights reserved.

Apple, the Apple logo, Apple IIGS, AppleLink, AppleShare, AppleTalk, APDA, APDAlog, GS/OS, HyperTalk, ImageWriter, LaserWriter, LocalTalk, Mac, MacAPP, Macintosh, MPW, MultiFinder and ProDOS are registered trademarks of Apple Computer, Inc. Develop, Finder, HyperMover, QuickDraw, and ViewEdit are trademarks of Apple Computer, Inc. Hypercard is a registered trademark of Apple Computer, Inc. and is licensed to Claris Corporation. ACOT is a service mark of Apple Computer, Inc. IBM is a registered tradmark of International Business Machines Corporation. MacDraw is a registered trademark of Claris Corporation. MacWeek is a registered trademark of Coast Associate Publishing, L.P. NuBus is a trademark of Texas Instruments. Postscript is a registered trademark of Adobe Systems Incorporated. Prototype is a trademark of Smethers Barnes.

LOUELLA PIZZUTI

Dear Readers,

Have you ever searched and searched for the answers to your questions only to find that they were right there under your nose the whole time? That's happened a lot with develop. We've made several changes in the one short year that we've been around: we've started printing the code dark enough so that you can read it, we're printing on recycled paper (see the letters section), we've added two columns, and we've trashed themes. The first two changes were direct results of your comments and our research (you see, we really do want to hear what you have to say), the other two were things I'm sure you thought of, but never got around to writing about.

The columns should help reduce the commitment that reading develop has become—we hope they'll provide you with some food for thought and with some time left over to ruminate.

And the themes, well, they just weren't working. Since the articles were only loosely grouped around a theme, you didn't get enough information to decide whether to read an issue or to pass it along (I most sincerely hope you'd never consider trashing it); it also forced us to defer some articles until there was a theme or an issue they fit in. So without themes, we'll let you decide how to categorize them (and still look forward to your article suggestions and ideas), and we'll concentrate on filling each issue with thorough, helpful, and interesting articles.

We're a nosey bunch and we want to know what you're thinking and what you're hoping. (Don't consider this a threat, but if we don't get more feedback, we'll have to do a survey to figure out what you want.)

So, nose around the articles and the code and pick out what interests you. Then let us know what intrigued you and what baffled you (we strive to intrigue, not baffle).

Louella Pizzuti
Editor

SUBSCRIPTION INFORMATION

Use the order form on the last page of the journal to subscribe to develop. Please address all subscription (and subscription-related) inquiries to develop, Apple Computer, Inc., P.O. Box 531, Mt. Morris, IL 61054 (AppleLink Dev.Subs).•

BACK ISSUES

Back issues of develop are available through APDA® (see inside back cover for APDA information), and are, of course, there for the browsing on each CD.•

LETTERS

Thanks for the excellent article in the October 1990 issue of develop dealing with polymorphism in C++ stand-alone code resources. I have one question about the code that accompanied it on the disc; it concerns the file WindowDef_main.cp: why are the overloaded "new" and "delete" operator definitions bracketed by the #ifdef NEEDED and #endif statements? Is NEEDED defined somewhere else? (I couldn't find it in any of the other files on the CD.) Are there circumstances in which you wouldn't want to overload the storage operators for a window definition function? I'm confused.

—Carlos Weber, M.D.

Yahoo, a technical buck to stop! Lemme at it!

The code in question (#ifdef NEEDED... #endif) shouldn't be there at all, and as a matter of fact is ignored by the compiler, since NEEDED isn't defined. It is left over from when Patrick was developing the code and still experimenting.

In the final version he is basing his WindowDefinition class on the Relocatable class, which is in turn based on HandleObject, an Apple® extension to C++ which uses handles instead of pointers when allocating space for new objects. No overloading of the storage operators is necessary.

Sorry about the confusion. I should have spotted that code and yanked it out before we published.

—*Dave Johnson*

In develop, Issue 4, you advocate installing DRVR resources at startup time by changing their resource ID to an empty slot in the Unit Table and then calling OpenDriver rather than use _DrvrInstall due to its bug.

The problem with this method is that it actually modifies the DRVR resource, which has two consequences: 1. According to Apple you are not supposed to modify yourself; and 2. it sets the last date modified field on the file. This latter problem causes backup programs to think the file has really changed and it worries users that perhaps some virus has modified that file when they know they didn't.

Thus, in my opinion, it is much better to use _DrvrInstall and, until Apple fixes the bug, stuff the DCE yourself. This method does not suffer the side effects of the OpenDriver method.

—Jeff Shulman

You're right, it's easier to use the method outlined in my article, but it's "better" to use _DrvrInstall and manually put the pointer to the driver into the appropriate field in the DCE.

—*Tim Enwall*

COMMENTS

We welcome timely letters to the editors, especially from readers wishing to react to articles that we publish in develop. Letters should be addressed to Dave Johnson or Louella Pizzuti at Apple Computer, Inc., Developer Programs, 20525 Mariani Ave., M/S 75-3B, Cupertino, CA 95014 (AppleLink: Johnson.DK or Pizzuti1).

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).•

LETTERS

I have a question about the article on the 8•24 GC card in develop, Issue 3, which, incidentally, was excellent. On pages 338 and 339 you mention the files that must be present to use the card. Is the GC file a transparent patching upgrade or is it a supplemental code block that duplicates all of the 32-Bit QD file functionality?

Good luck with Dogcow breeding,

—David

The 8•24 GC file contains more (and less) than just the 29000 equivalent of 32-Bit QuickDraw™; it also contains the IPC software that steals QD calls and transfers them to the card, the shell (GC OS) that receives the commands and dispatches them as well as doing the Memory Management chores and such, and finally the 'drawing' parts of 32 QD. Note that calls that do not cause any drawing to take place, such as NewGWorld, are not part of GC QD but are executed by the main processor even when acceleration is on.

All these functions are not part of 32 QD and therefore make it necessary to have the 8•24 GC file present when you want to have acceleration. So when running 6.0.x you need both files, 32 QuickDraw and 8•24 GC; under 7.0 you will need to have the 8•24 GC file present.

—Guillermo Ortiz

Have you guys considered adding perfume to the CD envelope, to try to raise a little capital from advertising to offset your costs? "C perfume, for the programmer in every man," "L'Air du Comp, as fresh as a new CPU."

—Jason Rusoff

My copy of develop arrived in my P.O. box this morning and therein lies the problem. In its original shape develop will not fit into my P.O. box so its shape was altered to make it fit. The CD is warped, and try as it might, the Finder™ cannot make the "minor repairs" it says are needed. Unfortunately, we out here on the frontier don't have someone to bring our mail to us each day; we have to go fetch it—and pay for the privilege. So I'm stuck with the U.S. Postal Service and an unusable CD.

Perhaps a large "DO NOT BEND" on the outside of subsequent issues will avert deformation. Then again, maybe not. Would you kindly use your considerable influence to get me a flat CD?

And if the printed warning doesn't work, how about a steel plate in each issue? Make that a really stiff steel plate. You know how those Postal Service people are—neither rain nor snow nor CD. . .

—Warren Michelsen

Sorry about that mangled CD—pushing the limits on information distribution sometimes we run into hassles like how to actually get the information into the hands (and drives) of the folks that need it. Hopefully our "DO NOT BEND" notice will help because I'd sure hate to have to resort to a steel plate.

If in fact, the warning doesn't help, you can contact the fulfillment house (see the subscription order form at the back of the issue for contact information) and tell them that your CD was mauled; they'll be happy to send you a new copy.

—Louella

In response to recent concern regarding the ecological soundness of your page layout, I have the following comments:

CD-ROM and electronic magazines are examples of technologies that—provided they are adopted—are ecologically superior to more traditional media, such as paper.

develop is encouraging the adoption of these new media, and thus can easily defend its spacious page layout. It is one of the few publications that can have a positive environmental impact—provided that developers absorb and act upon its contents.

Incorporating these new technologies into useful products is the first step toward their eventual widespread adoption. By keeping develop easy-to-read, you are encouraging this trend.

P.S. Providing an e-mail address for comments would also decrease paper usage!

—Bryn Dyment

Thanks for your words of encouragement. Pushing CD distribution is one way we're trying to get away from killing forests; using recycled paper is another. Our production manager, Hartley Lesser, found a paper that meets our quality standards, that doesn't use toxic chemicals for de-inking, and that's available in the quantities we require. This issue is the first one printed on the new paper—let us know what you think.

Also, we are now even more available electronically (although Dave's much more connected than I am), so if you'd rather send e-mail, feel free to use the addresses we provide.

—Louella

May it be known by you and your wonderfully talented crew how very much I appreciate your efforts on behalf of creating the wonderful, informative, interesting, entertaining, and otherwise "slick" magazine, develop. The opportunity to see what others are doing, who those others are, and perhaps to learn more than you would from *MacWeek*, but less than from *Inside Macintosh*, is indeed welcome. Add to this the fact that you include a CD-ROM and I am hard-pressed to even IMAGINE a more valuable offering. Great job. Thanks a zillion!!

—Lance Drake

I noticed that Apple is looking for a new editor-in-chief for the major publications. You're not going anywhere, are you?

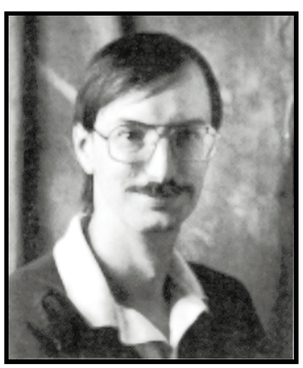
—A concerned reader

First of all, believe it or not, I did not make up this letter (or even the signature). Our group (Developer Technical Communications) recently reorganized and now my grouplet is responsible for not only develop, but also technical updates (like the Q & A stack), Technical Notes, Sample Code, and reporting compatibility bugs to developers. Since I need to spend my time doing the things that managers do, I'm looking for someone to do all the real work. If this sounds fun and you think you'd be qualified, let me know.

—Louella

ASYNCHRONOUS BACKGROUND NETWORKING ON THE MACINTOSH

LACS is a program that provides lightweight asynchronous conferencing for Macintosh® computers connected to the same AppleTalk® network. This article discusses the techniques used for implementing the asynchronous network operations, techniques that work well even when the application is running in the background under MultiFinder®. While the article provides the basic algorithms and techniques, the Developer Essentials *disc includes full source code for the entire LACS application.*



HARRY R. CHESLEY

Every Macintosh includes a local area network—LocalTalk®. Any two or more Macintosh computers can easily be configured to communicate with each other, passing data back and forth to work together as a larger system. There are many applications of computers that require or benefit from this sort of multiple workstation operation. Most of these applications involve groups of people working together and are known as collaborative computing, or computer supported cooperative work. While there are limited numbers of applications available in this category today, the numbers are increasing rapidly, and the potential for this genre is exciting.

Similarly, with the advent of MultiFinder and the ability to run programs in the background while the user continues working on a foreground application, it has become possible to write applications that operate on the user's behalf even when not immediately controlled by the user. These sorts of background "daemons" have long been available on mini, main-frame, and even workstation computers, but are relatively new to personal computers.

Network and background applications are, by their very nature, asynchronous. Network applications must communicate with other machines that may be slower than the local machine or busy with some other task. The other machines may even be temporarily turned off. Background operations must step very lightly to make sure that they don't affect the responsiveness of the system as perceived by the user. This usually involves using asynchronous techniques.

HARRY CHESLEY Due to a rare psychological impediment, Harry Chesley frequently finds himself incapable of giving short, simple answers to questions, instead reciting long stories that are only vaguely related to the original question. To spare you from having to read about Harry for hours, we took great care not to ask him too many questions about himself. Despite this cautious approach, we did discover that he has been at Apple so long that he no longer remembers his official title, that he has been programming the Macintosh since the 512K came out (and made it possible to do so without a Lisa), and that the first personal computer he bought was an Apple I (still buried in the closet somewhere). In the interest of brevity, the long stories behind each of these facts have been omitted. Even longer stories surround other events of his past life. He's been an

Asynchronous programs are often the hardest to design and develop. Our minds don't deal well with multi-threaded algorithms. And the Macintosh today has little in the way of development tools to help in this respect—there are no facilities for lightweight processes within Macintosh applications, for example.

The Lightweight Asynchronous Conferencing System (LACS) is a program that uses asynchronous background networking to propagate information from machine to machine. It distributes messages over a network of locally connected Macintosh computers. The LACS implementation provides examples of how to do the following:

- Invoke network operations asynchronously.
- Use an abstract superclass to simplify asynchronous design.
- Use NBP and ADSP.
- Operate in the background under MultiFinder.
- Implement a distributed database without requiring central control or coordination.

This article describes LACS, concentrating on the first two of these elements, with some limited discussion of the other items. You're encouraged to examine the source code of LACS (provided on the *Developer Essentials* disc) in order to uncover more details.

THE LIGHTWEIGHT ASYNCHRONOUS CONFERENCING SYSTEM (LACS) APPLICATION

LACS spreads messages from machine to machine across the local network. It is designed to run in the background under MultiFinder and communicate quietly with other machines. It uses the AppleTalk Name Binding Protocol (NBP) to find other machines to communicate with, and it uses the AppleTalk Data Stream Protocol (ADSP) to actually exchange messages. When new messages come in, the Notification Manager is used to alert the user.

LACS is written in Object Pascal using MacApp®. It uses object-oriented techniques to simplify the problem of implementing periodic asynchronous functions. To accomplish this, it uses an abstract superclass that provides a framework for other classes of the same type.

From the user's point of view, the application consists of three windows: Messages, New Message, and Status. Figure 1 shows these three windows. To create a new message, the user simply types in the New Message window and clicks the Send Message button. The message can be any text the user wants—but no pictures or graphics in this edition. The application then spreads the message to other locally connected Macintosh computers. When a new message arrives from another machine, it appears in the Messages window on that machine. The Messages

independent Mac software developer (the genesis of PackIt is a novella unto itself) and worked in a startup company named Metapath and at SRI (don't even ask). His favorite pastimes are playing with his two-and-a-half year old daughter (she has her own Macintosh but doesn't yet know how to run MPW®) and programming. Given the opportunity, he also enjoys writing about himself in the third person. •

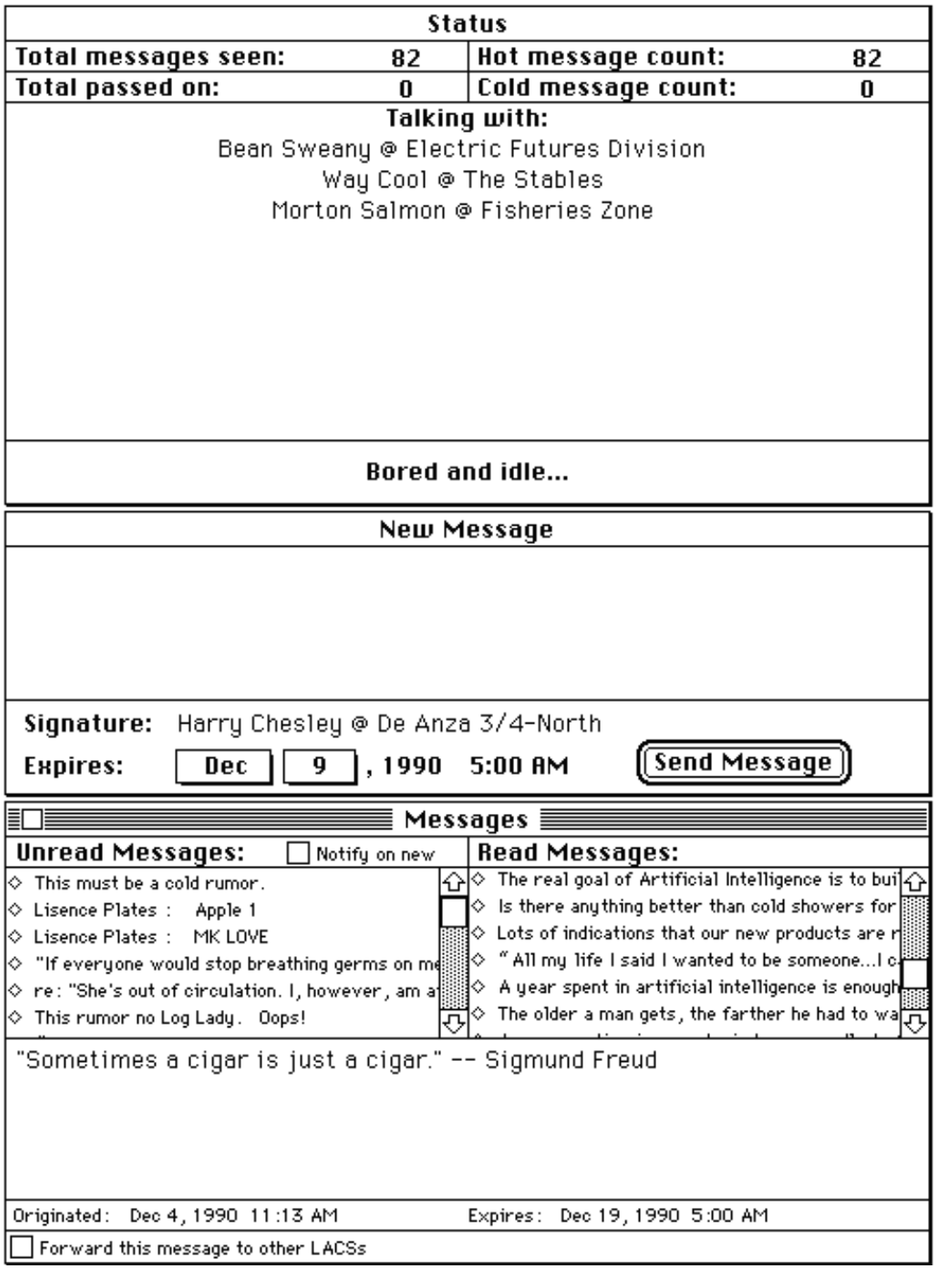


Figure 1
LACS User Interface

window displays two lists of messages, one for those which have yet to be read and one for those which the user has already read at least once. Only the first few words of each message appear in the read or unread list. When the user clicks on an entry in one of the lists, the full text of the message appears in a third section of the window. The Status window contains information about how many messages have been seen, what other machines are actively communicating, and so on.

There is more to the program than is covered in this brief description; for example, users can set expiration dates for the messages they create. You might want to run LACS to experience what it does and how it goes about it. However, the above description is sufficient for our purposes here. The basic operation and intent of the program is quite simple.

ALGORITHM FOR DISTRIBUTING MESSAGES

From the programmer's point of view, LACS maintains a distributed database of messages across multiple loosely connected computers. The central problem is how to distribute database updates across the network quickly and efficiently. The solution comes from a paper published by Xerox PARC: *Epidemic Algorithms for Replicated Database Maintenance.* (See references at the end of the article. Seems like everything interesting comes from PARC, doesn't it?) In fact, LACS was directly inspired by reading this paper.

In oversimplified form, the algorithm operates as follows: When a new message is first heard, it is considered "hot." The program then tries to tell other network nodes the new message. When a node passes on a message, the receiving node tells whether it's already heard the message or not. The more times the program tries to spread the message to nodes that have already heard it, the cooler the message becomes. Eventually it becomes completely cold and the program stops trying to spread the message to more nodes. The people at Xerox called the action of this algorithm "rumor mongering."

In LACS, the algorithm is implemented on top of the AppleTalk protocols. The Name Binding Protocol (NBP) is used to register LACS on the network. This allows the application to find other machines that are interested in exchanging messages. Each copy of LACS registers itself using the local machine's Chooser name with an NBP type of "LACS." The program then builds a list of other nodes of type "LACS." Rather than trying to maintain a list of all the systems on the net (potentially a very large list), it keeps up to ten nodes with which it communicates directly. These nodes communicate with up to ten others, they communicate with up to ten others, and so forth. Periodically, one of the entries in the local list is replaced with another machine chosen at random, so that the list slowly changes over time.

The Apple Data Stream Protocol (ADSP) is used to communicate between LACS systems. This protocol provides reliable byte-stream connections, correcting for any errors in transmission across the network. When a LACS machine decides to spread a message, it makes an ADSP connection with another LACS node. It exchanges messages with the other machine and then closes the ADSP connection.

MESSAGE EXCHANGE PROTOCOL

LACS implements a message exchange protocol on top of ADSP's reliable byte stream. This message exchange protocol consists of separate commands, each having a command name and a series of parameters. For example, the "Here's a new message" command includes the message itself, its origination and expiration dates, and other related information as parameters.

The ADSP session consists of a series of command exchanges. The originating node starts the conversation. The destination node then responds with a command of its own. Usually, the conversation starts with an attempt by the originator to pass on a message; this is known as "pushing." But under some circumstances, the originator may instead ask the other machine for a message; this is known as "pulling." In that case, the other machine takes control of the conversation and sends a message. When the controlling side has nothing further to say, the connection is closed.

The message exchange protocol commands include

- "Here's a new message."
  Valid responses: "I've seen it." or "I haven't seen it."
- "Give me a hot message."
  Valid responses: "Here's a new message."
- "Give me a message; I don't care if it's hot or cold."
  Valid responses: "Here's a new message."

The responses "I've seen it" and "I haven't seen it" are actually implemented as commands as well. But they are only generated in response to a "Here's a new message" command.

The protocol is very simple and is designed to use only ASCII text in the commands and responses. This makes it easy for someone to write a program other than LACS that can become part of the community of message spreaders. For example, a gateway could spread messages from the local network to a wider area network. Or an archive agent could collect and save messages.

Internally, LACS keeps track of the number of times it successfully or unsuccessfully tried to pass on a message. The number of failed attempts is used to determine when a message becomes cold and also how long to wait until the next attempt to pass it on.

The program actually implements several variations of the basic algorithm, which can be selected by changing a few global parameters to the program. The default parameters are

- Cool off messages deterministically (as opposed to stochastically).
- Consider a message to be cold after 30 redistribution failures. This makes it highly likely that all machines will see each message, since each machine tries 30 times to redistribute it. Of course, once most machines have seen the message, most redistribution attempts will fail, since they will more than likely pick a machine that has already seen the message.
- Push messages (rather than pull). This means that connections are only made when there actually are messages to be transmitted.
- When picking another LACS machine with which to communicate, look in the local AppleTalk zone twice as frequently as in other zones. This tends to reduce network overhead by keeping communications local.
- Use an exponential back-off when determining how long to wait before attempting to distribute the message again. This allows for quick initial redistribution, but keeps the messages "hot" for some time, so that they get to machines that were turned off when the message initially entered the network.

See the paper mentioned earlier from Xerox PARC, the source code of LACS, and the file "About LACS" on the *Developer Essentials* disc for complete details of the algorithm and variations used in LACS.

ASYNCHRONOUS OPERATION: TPERIODIC

LACS requires that several activities proceed asynchronously. Since it runs in the background under MultiFinder, it cannot wait for the completion of a network operation. It has to release control to the foreground process as quickly as possible. In addition, there are several semi-independent activities in the program. Making them dependent on each other, even to the extent that only one operates at a time, would unnecessarily complicate the design.

The semi-independent, asynchronous activities in LACS include the following:

- Build and maintain a list of AppleTalk zones.
- Build and maintain a list of other LACS nodes with which to communicate.
- Initiate outgoing communication sessions.
- Receive incoming communication sessions.
- Perform housecleaning operations, such as saving the message database to disk periodically.

In order to keep the design manageable, it is important to be able to separate these activities into distinct code modules. Each individual piece is relatively easy to understand and implement. It's only when they're taken together that the problem becomes difficult.

This design separation is provided by building an abstract superclass, TPeriodic, that implements periodic asynchronous operations. The model for TPeriodic is that asynchronous periodic activities follow a particular pattern:

[1] Wait for a set period of time.
[2] Initiate an asynchronous action.
[3] Check repeatedly to see if the action has completed.
[4] Do something with the result.
[5] Repeat from step 1.

The concrete subclasses of TPeriodic include TZoneLookup, TNodeLookup, TGossip, and TDocumentSaver. Each of these subclasses are discussed in more detail in subsequent sections.

INTERFACE

The interface to TPeriodic looks like this:

```
PeriodicStates = (kPeriodicInactive, kPeriodicWaiting, kPeriodicActive);

TPeriodic = object(TEvtHandler)
      fInactiveIdle: longInt;       { Idle period when inactive. }
      fActiveIdle: longInt;         { Idle period when waiting for completion. }
      fState: PeriodicStates;       { Current state. }

      procedure TPeriodic.IPeriodic(initialIdle, inactiveIdle,
                                    activeIdle: longInt);
            { Initialize the periodic object. }

      procedure TPeriodic.Free; override;
            { Free the periodic object. }

      procedure TPeriodic.WaitForAsync;
            { Wait until any asynchronous activity is finished. }

      procedure TPeriodic.Kick;
            { Get things moving right now. }

      function TPeriodic.DoIdle(phase: IdlePhase): boolean; override;
            { Internal method -- perform idle activities. }
```

```
procedure TPeriodic.Activate;
      { Start asynchronous operation.
            To be overridden by subclass. }

procedure TPeriodic.Waiting;
      { Check for completion of asynchronous operation.
            To be overridden by subclass. }

procedure TPeriodic.DoIt;
      { Take action after completion of asynchronous operation.
            To be overridden by subclass. }

end;
```

STATE MACHINE

TPeriodic implements the state machine shown in Figure 2.

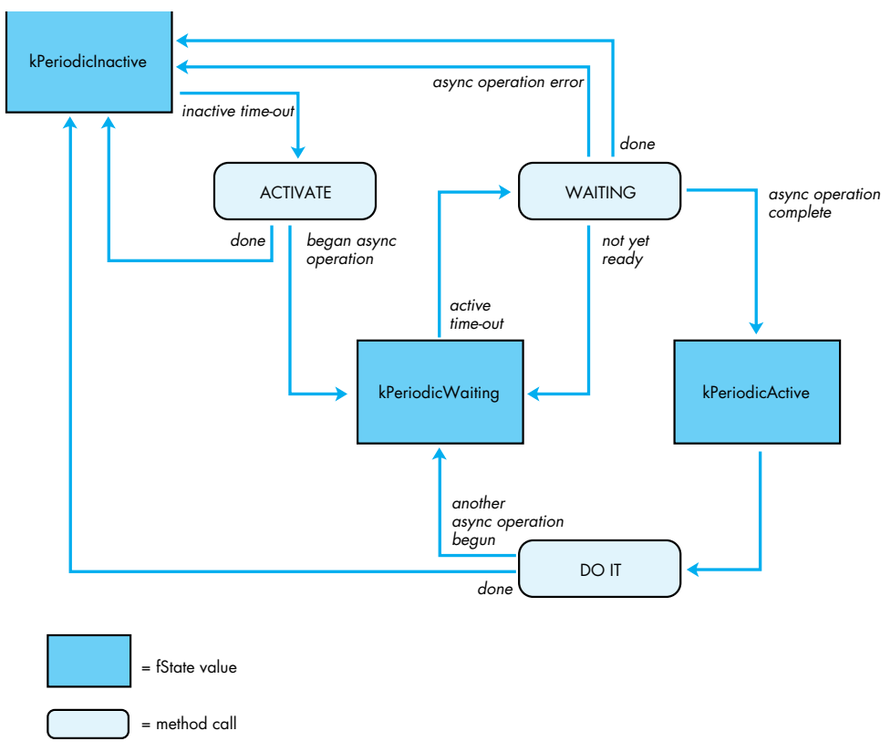


Figure 2
TPeriodic State Machine

IPeriodic and Free are called to initialize and free instances of TPeriodic. Kick is called to start an activity even though the time-out hasn't expired yet. Each subclass of TPeriodic overrides Activate, Waiting, and DoIt to implement their own functionality. DoIdle is an internal routine that is discussed later with the implementation. WaitForAsync is a utility method used by Free to wait until any outstanding asynchronous operations have completed before freeing the object.

The current state is stored in fState. While a TPeriodic object is waiting for the time-out to occur, fState is kPeriodicInactive. When the time-out happens, Activate is called, which sets fState to some new value. If the Activate procedure has started an asynchronous activity, it sets fState to kPeriodicWaiting. If it has taken some synchronous action and then wants to go back to sleep, it sets fState to kPeriodicInactive again. While in kPeriodicWaiting, the method Waiting is called repeatedly. Waiting's task is to test for completion of the asynchronous activity begun by Activate. When completion occurs, it sets fState to kPeriodicActive. If no action is needed after the asynchronous activity, or if the asynchronous activity failed, Waiting sets fState back to kPeriodicInactive. Finally, when fState is kPeriodicActive, DoIt is called immediately (no time delay). The DoIt method takes the appropriate actions with the results of the asynchronous activity, and then sets fState to kPeriodicInactive. The process then repeats.

The instance variable fInactiveIdle determines the length of time between activations. The variable fActiveIdle determines the length of time between calls to Waiting when an asynchronous activity has been started—that is, how often to check if it's finished yet. These are set by the inactiveIdle and activeIdle parameters to IPeriodic. The initialIdle parameter to IPeriodic determines the initial time-out to be used—how long to wait for the very first activation.

TPeriodic is a subclass of the MacApp TEvtHandler class. IPeriodic installs the instance in the MacApp cohandler chain. Most of the work is then done by facilities already built into MacApp. DoIdle, which is called by MacApp software when the cohandler's time-out occurs, simply decides which of the Activate, Waiting, and DoIt methods to call. Those methods set fState to determine what to do next. Since Activate, Waiting, and DoIt are implemented by the subclasses, TPeriodic consists only of the four methods IPeriodic, Free, DoIdle, and Kick plus one internal utility method, WaitForAsync. Here is the actual code:

```
procedure TPeriodic.IPeriodic(initialIdle, inactiveIdle, activeIdle: longInt);
      { Initialize the object. }

      begin
            IEvtHandler(nil);
            fIdleFreq := initialIdle;
            fInactiveIdle := inactiveIdle;
            fActiveIdle := activeIdle;
            fState := kPeriodicInactive;
```

```
            { Install the object in the co-handler chain. }
            gApplication.InstallCohandler(self,true);
      end;

procedure TPeriodic.Free;
      { Free the object. }

      begin
            { First wait for any outstanding operation to complete. }
            WaitForAsync;
            { Deinstall ourselves from the co-handler chain. }
            gApplication.InstallCohandler(self,false);
            { Free ourselves. }
            inherited Free;
      end;

procedure TPeriodic.WaitForAsync;
      { Wait until any asynchronous activity is finished. }

      begin
            while fState = kPeriodicWaiting do Waiting;
      end;

function TPeriodic.DoIdle(phase: IdlePhase): boolean;
      { Internal method -- idle the object. }

      var fi: FailInfo;

      procedure hdlFailure(error: OSErr; message: LongInt);
            { If we fail, reset to inactive. }

            begin
                  fState := kPeriodicInactive;
                  fIdleFreq := fInactiveIdle;
                  exit(DoIdle);
            end;

      begin
            DoIdle := false;
            if phase = IdleContinue then
                  begin
                        CatchFailures(fi,hdlFailure);
                        { If we've just timed out, then activate the object. }
                        if fState = kPeriodicInactive then Activate
                        else
```

```
                        begin
                           { If we're waiting, see if we're done yet. }
                           if fState = kPeriodicWaiting then Waiting;
                           { If we're done, do something with the results. }
                           if fState = kPeriodicActive then DoIt;
                        end;
                  { Figure out the new idle frequency. }
                  if fState = kPeriodicInactive then fIdleFreq :=
                                                      fInactiveIdle
                  else fIdleFreq := fActiveIdle;
                  Success(fi);
            end;
      end;

procedure TPeriodic.Kick;
      { Start things up even if it isn't normally time yet. }

      begin
            fIdleFreq := 0;
      end;
```

MAINTAINING AN APPLETALK ZONES LIST: TZONELOOKUP

In order to maintain its list of AppleTalk zones, LACS contains a TPeriodic subclass called TZoneLookup. Looking up the list of AppleTalk zones can be done in one of two ways. The old way involved talking directly to a nearby AppleTalk Router. The newer and simpler way, which is used in LACS, makes use of AppleTalk Phase 2 calls, leaving the underlying communication with the Router to the AppleTalk software.

However, the new technique will not work with older versions of AppleTalk (prior to System 6.0.3). It is important, therefore, to check and make sure that AppleTalk Phase 2 is available before using TZoneLookup. This test is performed in TLACSApplication.ILACSApplication, and takes the following form:

```
{ Check for AppleTalk phase 2. }
if gConfiguration.atDrvrVersNum < 53 then
  begin
        StdAlert(phNoPhase2);
        ExitMacApp;
  end;
```

The interface to TZoneLookup primarily overrides the TPeriodic methods Activate, Waiting, and DoIt. In addition, it defines several constants, a data type used in the new AppleTalk calls, instance variables used by TZoneLookup, and the initialization function IZoneLookup.

```
const

{ csCodes for new .XPP driver calls: }
xCall = 246;

{ xppSubCodes: }
zipGetLocalZones = 5;
zipGetZoneList = 6;
zipGetMyZone = 7;

type

{ Offsets for xCall queue elements: }
xCallParam =
      packed record
            qLink: QElemPtr;
            qType: INTEGER;
            ioTrap: INTEGER;
            ioCmdAddr: Ptr;
            ioCompletion: ProcPtr;
            ioResult: OsErr;
            ioNamePtr: StringPtr;
            ioVRefNum: INTEGER;
            ioRefNum: INTEGER;
            csCode: INTEGER;
            xppSubCode: INTEGER;
            xppTimeOut: Byte;
            xppRetry: Byte;
            filler: INTEGER;
            zipBuffPtr: Ptr;
            zipNumZones: INTEGER;
            zipLastFlag: INTEGER;
            zipInfoField: packed array[1..70] of Byte;
      end;

xCallPtr = ^xCallParam;

const

kXPPTimeOutVal = 3;                 { Re-try XPP attempt every 3 seconds. }
kXPPRetryCount = 5;                 { For five times. }
kZonesBufferSize = 578;             { Size of buffer for zone names. }
kMaxZones = 100;                    { Maximum number of zones to handle. }

type
```

```
TZoneLookup = object(TPeriodic)
      fDocument: TLACSDocument;           { The document we're looking up for. }
      fZoneCount: integer;                { How many zones we've found. }
      fXPPPBPtr: xCallPtr;                { XPP parameter block. }
      fZonesBuffer: Ptr;                  { Input buffer. }
      fOurZone: Str32;                    { The name of our own zone. }
      fZones: array [1..kMaxZones] of Str32;    { Zone names. }

      procedure TZoneLookup.IZoneLookup(aDoc: TLACSDocument;
                              initialIdle, inactiveIdle, activeIdle: longInt);
            { Initialize the zone lookup object. }

      procedure TZoneLookup.Free; override;
            { Free the zone lookup object. }

      procedure TZoneLookup.Activate; override;
            { Start a zone list lookup. }

      procedure TZoneLookup.Waiting; override;
            { Wait for the zone lookup to complete. }

      procedure TZoneLookup.DoIt; override;
            { Process returned zone list. }

      end;
```

INITIALIZATION AND FREEING

During initialization, in IZoneLookup, the current zone list size is cleared out, emptying the list, and some buffers that are needed for the zone lookup procedure are allocated:

```
procedure TZoneLookup.IZoneLookup(aDoc: TLACSDocument;
                        initialIdle, inactiveIdle, activeIdle: longInt);
     { Initialize the zone lookup object. }

     begin
            IPeriodic(initialIdle,inactiveIdle,activeIdle);
            fDocument := aDoc;
            fOurZone := '';
            fZoneCount := 0;
            { Allocate memory blocks we'll need later. }
            fXPPPBPtr := xCallPtr(NewPtr(sizeof(xCallParam)));
            FailNil(fXPPPBPtr);
            fZonesBuffer := NewPtr(kZonesBufferSize);
            FailNil(fZonesBuffer);
     end;
```

When the TZoneLookup object is freed, it waits for any asynchronous activity to complete and then deallocates its buffers. If it didn't do the WaitForAsync call, an outstanding operation might try to write something into one of the buffers after it was deallocated.

```
procedure TZoneLookup.Free;
      { Free the zone lookup object. }

      begin
            WaitForAsync;
            DisposPtr(Ptr(fXPPPBPtr));
            DisposPtr(fZonesBuffer);
            inherited Free;
      end;
```

ZONES LIST REQUEST

When the zone lookup process is started in Activate, two actions are taken. First, the local zone name is obtained and stored for future use. This is done synchronously, since it shouldn't take very long. Second, an asynchronous request for a complete list of zones is issued. The results of the request will be dealt with in DoIt.

```
procedure TZoneLookup.Activate;
      { Start a zone list lookup. }

      var addrBlock: AddrBlock;
            ignore: integer;
            s: Str255;

      begin
            { Let the user know what we're doing. }
            fDocument.fStatusWindow.SetStatus(kStatZoneUpdate);
            { Clear out the zone list. }
            fZoneCount := 0;
            { Get our zone name. }
            with fXPPPBPtr^ do
                  begin
                        ioRefNum := xppRefNum;        { Driver refNum -41. }
                        csCode := xCall;
                        xppSubCode := zipGetMyZone;
                        zipBuffPtr := @s;
                        zipInfoField[1] := 0;         { ALWAYS 0. }
                        zipInfoField[2] := 0;         { ALWAYS 0. }
                  end;
            { Send the getMyZone request synchronously (and cross }
            {     our electronic fingers it doesn't take long). }
```

```
            if PBControl(ParmBlkPtr(fXPPPBPtr), false) <> noErr then
                  fState := kPeriodicInactive
            else
                  begin
                        { Update the display to reflect any changes. }
                        if (s <> fOurZone) and (s <> '') then
                              begin
                                 fOurZone := s;
                                 fDocument.fNewWindow.GetSignature;
                              end;
                        { Now make a getZoneList request. }
                        with fXPPPBPtr^ do
                              begin
                                 { ALWAYS 0 on first call; contains state info on
                                 {  subsequent calls. }
                                 zipInfoField[1] := 0;
                                 { ALWAYS 0 on first call; contains state info on
                                 {  subsequent calls. }
                                 zipInfoField[2] := 0;
                                 ioRefNum := XPPRefNum;{ Driver refNum -41. }
                                 csCode := xCall;
                                 xppSubCode := zipGetZoneList;
                                 xppTimeOut := kXPPTimeOutVal;
                                 xppRetry := kXPPRetryCount;
                                 { This buffer will be filled with packed zone
                                 {  names. }
                                 zipBuffPtr := Ptr(fZonesBuffer);
                                 zipLastFlag := 0;
                              end;
                        { Send off the request. }
                        ignore := PBControl(ParmBlkPtr(fXPPPBPtr), true);
                        fState := kPeriodicWaiting;
                  end;
      end;
```

PERIODIC CHECKING

The Waiting method then checks periodically to see if the result has come in or an error has occurred.

```
procedure TZoneLookup.Waiting;
      { Wait for the zone lookup to complete. }

      begin
            if fXPPPBPtr^.ioResult = noErr then fState := kPeriodicActive
            else if fXPPPBPtr^.ioResult < noErr then fState := kPeriodicInactive;
      end;
```

COLLECTING THE RESULTS

Finally, when the result is available, DoIt is called to record the new zones in the local list. If there were more zones than fit in this message, another asynchronous call is made, and the state returns to kPeriodicWaiting. Otherwise, the zone lookup process is finished.

```
procedure TZoneLookup.DoIt;
      { Process returned zone list. }

      var dCount: integer;
            dCurr: Ptr;
            ignore: OSErr;

      begin
            { Cycle through the returned list. }
            dCount := fXPPPBPtr^.zipNumZones;   { Find out how many returned. }
            dCurr := fZonesBuffer;              { Put current pointer at start. }
            while (fZoneCount < kMaxZones) and (dCount > 0) do { Get each zone. }
                  begin
                        fZoneCount := fZoneCount+1;
                        fZones[fZoneCount][0] := chr(dCurr^);
                        BlockMove(pointer(ord4(dCurr)+1),
                              pointer(ord4(@fZones[fZoneCount])+1),dCurr^);
                        dCurr := pointer(ord4(dCurr) + dCurr^+1);
                        dCount := dCount-1;
                  end;
            { If there are more to come, do another request. }
            if (fZoneCount < kMaxZones) and (fXPPPBPtr^.zipLastFlag = 0) then
                  begin
                        ignore := PBControl(ParmBlkPtr(fXPPPBPtr), true);
                        fState := kPeriodicWaiting;
                  end
            { Otherwise, we're all done. }
            else fState := kPeriodicInactive;
      end;
```

LOOKING UP NODES: TNODELOOKUP

Finding the NBP names of other LACS systems is handled by class TNodeLookup. This class uses standard NBP name lookup procedures, and is otherwise similar to TZoneLookup. Therefore, its implementation is left as an exercise for the reader (or for the CD-ROM driver, as the full source code can be found on the *Developer Essentials* disc). Meanwhile, we will jump straight into the TGossip class.

MESSAGE PASSING: TGOSSIP

Being a gossiper (who initiates the communication session) and being a gossipee (who listens for others who want to communicate with it) are much the same, so they are implemented as the same class, but with a parameter to IGossip to declare which case a particular instance is. Two copies of TGossip are instantiated, one to initiate message-passing sessions over ADSP and one to respond. The TGossip class looks much the same as the other TPeriodic subclasses:

```
TGossip = object(TPeriodic)
      fDocument: TLACSDocument;          { The document we're communicating in. }
      fOutgoing: boolean;                { Whether this is an outgoing gossiper. }
      fDidPull: boolean;                 { Whether we just did a pull. }
      fADSPSocket: integer;              { Our socket number. }
      fADSP: DSPPBPtr;                   { The ADSP IO block pointer. }
      fCcbPtr: Ptr;                      { CCB for ADSP. }
      fSendQueue: Ptr;                   { Send queue for ADSP. }
      fRecvQueue: Ptr;                   { Receive queue for ADSP. }
      fAttnPtr: Ptr;                     { Attention pointer for ADSP. }
      fADSPData: Ptr;                    { The data buffer pointer. }
      fNTE: ^NamesTableEntry;            { Our names table entry. }


procedure TGossip.IGossip(aDoc: TLACSDocument; outgoing: boolean;
                  initialIdle, inactiveIdle, activeIdle: longInt);
      { Initialize the gossip object. }

procedure TGossip.Free; override;
      { Free the gossip object. }

procedure TGossip.Activate; override;
      { Start a new gossip session (outgoing only). }

procedure TGossip.Waiting; override;
      { Wait for more input or a connection to open. }

procedure TGossip.DoIt; override;
      { Handle new input. }

procedure TGossip.PassiveOpen;
      { Do a passive connection open. }

procedure TGossip.ResetConnection;
      { Reset the connection. }

      end;
```

We'll quickly summarize the straightforward methods of TGossip, and go into detail only on the central DoIt method. Again, full details can be found in the source code on the *Developer Essentials* disc.

INITIALIZATION

IGossip allocates buffers and prepares for connections. If this is for incoming messages (the outgoing parameter is false), IGossip starts up a listen for a new ADSP connection (by calling PassiveOpen) and registers the system's name using NBP. If this object is going to be initiating connections (the outgoing parameter is true), it simply waits for the first Activate time-out to take action.

INITIATING A CONNECTION

The Activate method is used to initiate a connection with another LACS machine. The other LACS machine is chosen at random and an ADSP session initiated by issuing an ADSP active open call. After the connection is open, the rest of the work is done in DoIt. Note that Activate is never called for instances that are waiting for incoming connections—they always go straight to the kPeriodicWaiting state since they're always waiting for another connection from outside.

TESTING FOR COMPLETION

The method Waiting tests for the completion of the last ADSP request, the result of which is either an opened connection or a completed data transmission. Whether the connection was initiated by this object or the other communicating system depends upon whether this is an incoming or outgoing gossip object. And a data transmission may be either a send or a receive. (A discussion of how data transmissions get started comes later.)

THE HEART OF TGOSSIP: DOIT

DoIt is called when a network operation has completed. Initially, that operation is the establishment of a connection. Once the connection is established, the machine that initiated it needs to generate the first command to be sent; the machine that did not initiate the connection needs to start up a receive to obtain that command from the other node. Once the connection is open and a command is sent and received, a response must be constructed and sent. All of this is done by DoIt.

While this seems like a lot of functionality to cram into one routine, it isn't really all that bad. Because there are no distractions from other aspects of the communication activity, and because much of the component functionality is provided by other parts of the system (by TMessage, for example), all the functionality can be included in one routine without overloading the programmer who is reading, writing, or maintaining that code.

The following is a slightly simplified copy of TGossip.DoIt. (See the source code on the *Developer Essentials* disc for the full implementation.) DoIt decides which of the possible operations it should perform based on the csCode field of the I/O block.

This field tells what operation was last requested (open, read, or write). The csCode field is effectively used as another state machine within the state machine already in use and defined by fState. (State machines tend to be very useful in implementing asynchronous algorithms.)

If the connection just opened, the initiator must find a hot message to spread, build a "Here's a new message" command, and start an ADSP write. After the receiving node reads the command from the connection, the command is passed to HandleIncomingCommand. That routine builds a reply command, which needs to be sent via another write. If a write or a non-initiated opening of a connection just occurred, the receiver starts up an ADSP read.

Most of the real work is done in GetHotMessage, BuildMessage, and HandleIncomingCommand. GetHotMessage decides what messages to send. BuildMessage and HandleIncomingCommand implement the message exchange protocol on top of ADSP. These methods are implemented in other classes of the system, which know more about those other parts of the application. For example, BuildMessage is in the class TMessage, which knows all the internal details of a message object.

```
procedure TGossip.DoIt;
     { Handle new input. }

     var r: TMessage;
          p: Ptr;
          noGood: boolean;

     begin
          noGood := false;
          { If this is a session open and we're the initiator... }
          if (fADSP^.csCode = dspOpen) and fOutgoing then
               begin
                    { Get a message to send. }
                    r := fDocument.GetHotMessage;
                    { Decide if we've something to send. }
                    if r <> nil then
                         begin
                              { Generate the appropriate send request. }
                              with fADSP^ do
                                   begin
                                        p := fADSPData;
                                        reqCount := r.BuildMessageCommand(p);
                                        dataPtr := fADSPData;
                                        eom := 1;
                                        flush := 1;
                                        csCode := dspWrite;
```

```
                              end;
                         { Send it. }
                         if PBControl(ParmBlkPtr(fADSP),true) <> noErr then
                              noGood := true;
                    end
               else noGood := true;
          end
     { If this is a completed read... }
     else if fADSP^.csCode = dspRead then
          begin
          { Handle the incoming command, and build a reply if appropriate. }
               with fADSP^ do
                    begin
                         reqCount := fDocument.HandleIncomingCommand
                                        (fADSPData,fADSP^.actCount);
                         dataPtr := fADSPData;
                         eom := 1;
                         flush := 1;
                         csCode := dspWrite;
                    end;
               { If there's a reply, send it. }
               if fADSP^.reqCount > 0 then
                    begin
                         if PBControl(ParmBlkPtr(fADSP),true) <> noErr then
                              noGood := true;
                    end
               else noGood := true;
          end
     { Otherwise... }
     else
          begin
               { Start up a receive. }
               with fADSP^ do
                    begin
                         dataPtr := fADSPData;
                         reqCount := kADSPMaxCommand;
                         csCode := dspRead;
                    end;
               if PBControl(ParmBlkPtr(fADSP),true) <> noErr then
                    noGood := true;
          end;
     { If we're all done, reset the connection. }
     if noGood then ResetConnection
     { Otherwise, wait for the results. }
     else fState := kPeriodicWaiting;
end;
```

AUTO-SAVING: TDOCUMENTSAVER

It's also possible to use TPeriodic for activities which are not related to the network at all, for example, to automatically save a document periodically. Since LACS is intended to be kept running all the time the Macintosh is on, it can accumulate a large number of changes to its message database over time. The user could periodically issue a command to save the data base to disk, but it's much nicer if LACS does it automatically. TDocumentSaver provides that functionality.

When the time-out occurs, TDocumentSaver waits for LACS to be in the foreground and then saves the document to disk. Saving could occur in Activate, without waiting for the application to be in the foreground. But saving can potentially take several seconds, much too long an activity for a background task. On the other hand, the document could be saved asynchronously, one piece at a time, in the background. But that would have been difficult to implement, since none of the MacApp existing document read/write structure could be used. It uses an entirely synchronous implementation.

```
procedure TDocumentSaver.IDocumentSaver(aDoc: TLACSDocument;
                              initialIdle, inactiveIdle, activeIdle: longInt);

      begin
            IPeriodic(initialIdle,inactiveIdle,activeIdle);
            fDocument := aDoc;
      end;

procedure TDocumentSaver.Activate;

      begin
            fState := kPeriodicWaiting;
      end;

procedure TDocumentSaver.Waiting;

      begin
            if not gInBackground then fState := kPeriodicActive;
      end;

procedure TDocumentSaver.DoIt;

      begin
            fDocument.Save(cSave,false,false);
            fState := kPeriodicInactive;
      end;
```

INITIALIZING AND LAUNCHING THE PERIODIC OBJECTS

Each of the periodic objects must be allocated and initialized. In LACS, this happens in the TLACSDocument initialization methods as follows:

```
const

{ Document saver: }
kDocSaverInitial = 60*60*30;              { 30 minutes. }
kDocSaverInactive = 60*60*30;             { 30 minutes. }
kDocSaverActive = 30;                     { 1/2 second. }

{ Zone lookup: }
kZoneLookupInitial = 0;                   { Right away. }
kZoneLookupInactive = 60*60*60*4;         { 4 hours. }
kZoneLookupActive = 30;                   { 1/2 second. }

{ Node lookup: }
kNodeLookupInitial = 60*8;                { 8 seconds. }
kNodeLookupFastInactive = 60*8;           { 8 seconds. }
kNodeLookupSlowInactive = 60*60*20;       { 20 minutes. }
kNodeLookupActive = 30;                   { 1/2 second. }

{ Gossipee: }
kGossipeeInitial = 0;                     { Right away. }
kGossipeeInactive = 60*60 + 13;           { 1 minute. }
kGossipeeActive = 30;                     { 1/2 second. }

{ Gossiper: }
kGossiperInitial = 60*21;                 { 21 seconds. }
kGossiperInactive = 60*30 + 27;           { 30 seconds. }
kGossiperActive = 30;                     { 1/2 second. }

var ds: TDocumentSaver;
     zl: TZoneLookup;
     nl: TNodeLookup;
     g: TGossip;

.
.
.
```

```
{ Document saver. }
new(ds);
FailNil(ds);
ds.IDocumentSaver(self,kDocSaverInitial,kDocSaverInactive,
                        kDocSaverActive);
fDocumentSaver := ds;

{ Zone lookup. }
new(zl);
FailNil(zl);
zl.IZoneLookup(self,kZoneLookupInitial,kZoneLookupInactive,
                        kZoneLookupActive);
fZoneLooker := zl;

{ Node lookup. }
new(nl);
FailNil(nl);
nl.INodeLookup(self,kNodeLookupInitial,kNodeLookupFastInactive,
                        kNodeLookupSlowInactive,kNodeLookupActive);
fNodeLooker := nl;

{ Gossipee. }
new(g);
FailNil(g);
g.IGossip(self,false,kGossipeeInitial,kGossipeeInactive,
                        kGossipeeActive);
fGossipee := g;

{ Gossiper. }
new(g);
FailNil(g);
g.IGossip(self,true,kGossiperInitial,kGossiperInactive,
                        kGossiperActive);
fGossiper := g;
```

Note that some of the idle times are slightly odd numbers. This is to keep activities from becoming synchronized—occurring at the same time—and therefore taking a noticeable amount of processing time within a particular time period.

USER INTERFACE AND THE INTERNAL DATABASE

The rest of LACS is concerned with the internal message database and with the user interface.

The internal database consists of a collection of objects of class TMessage. Each of these objects contains a message and includes the text of the message, the date it was created, the date it is to expire, how many times it has successfully been passed on to other LACS systems, how many times it was unsuccessfully passed on because the recipient had already heard it, and so forth. The messages are kept in a TLACSDocument object. Besides holding the message objects, TLACSDocument knows how to search for hot messages.

The user interface is handled by vanilla MacApp classes such as TStaticText, TCheckBox, TTEView, and so on. The only special case is in handling the read and unread message lists. For this, LACS creates the subclasses TTextListView and TSortedList to provide a new pair of classes that know about each other and automatically propagate changes between the two paired objects. When a new object is inserted in TSortedList, it is immediately added to the TTextListView in the Messages window.

All of these objects are managed by TLACSDocument, which acts as a central coordinator for the database and message-passing activities of the system. For example, TLACSDocument includes HandleIncomingCommand, which decides what actions to take based on an incoming command from an ADSP connection. As currently implemented, there is only one TLACSDocument active at a time. However, by combining the active elements of the database and network into a document object, it is an easy extension to allow multiple simultaneous databases to be active. This allows for the possibility of parallel sets of LACS systems divided by topic or security level. It also opens the door to other types of documents supported by the same application—archiving or gateway functions, for instance.

SUMMARY

The Lightweight Asynchronous Conferencing System (LACS) implements a distributed database update algorithm in order to spread messages around a local network using AppleTalk protocols. It is implemented in Object Pascal using MacApp. In order to accomplish its goals, LACS must implement multiple asynchronous background activities.

Asynchronous network and background operations tend to be challenging to implement on the Macintosh. Much of the difficulty involves the inherent problems of dealing with parallel algorithms—we humans prefer to deal with one thing at a time.

It is possible, however, to greatly reduce the cognitive burden of implementing this type of algorithm by providing a proper context for the implementation. LACS does this by creating an abstract superclass in the MacApp environment. Using this approach, the problem is isolated from other, irrelevant, parts of the application, and the individual parts of the particular asynchronous activity are clearly broken out.

This abstract superclass is called TPeriodic. It implements the generic algorithm "wait for time out; start asynchronous operation; wait for asynchronous operation to complete; do something with the results; start over." The algorithm is implemented as a state machine, driven by TPeriodic, but with details supplied by TPeriodic's subclasses. Specific subclasses may use all or only portions of the state machine.

Using the TPeriodic framework, it becomes quite straightforward to implement classes for performing zone name lookup, node name lookup, initiating messages, receiving messages, automatically saving the document to disk, and more. Each individual function is implemented separately without regard for the other periodic, asynchronous functions in the system.

In addition to the explanation of the TPeriodic class within this article, it is hoped that the source code supplied on CD-ROM will serve as an example of how to implement each of the separate pieces of network functionality listed. Many of them were taken from example code fragments in the Technical Notes, Sample Code, and elsewhere, but they are drawn together here into a coherent, working application and integrated into the MacApp environment.

ACKNOWLEDGMENTS

This program would be roughly half as good as it is if it weren't for Brian Bechtel. He provided suggestions, encouragement, testing, evangelism, and the ear the Notification Manager flashes in the menu bar. Thanks also to Michael Gough, who pointed me to the article from Xerox PARC, which got me started on this whole project in the first place. And thanks to our early group of testers, who put up with a fair number of flaky releases. Of course, all of this was done in the very serious pursuit of collaborative computing research.

For information on collaborative computing see

Irene Greif: *Computer-Supported Cooperative Work: A Book of Readings*, Margaret Kaufmann Publishers, Inc., 1988.

For details on the algorithm see

Alan Demers, Mark Gealy, Dan Greene, Carl Hauser, Wes Irish, John Larson, Sue Manning, Scott Shenker, Howard Sturgis, Dan Swinehart, Doug Terry, Don Woods: *Epidemic Algorithms for Replicated Database Maintenance*, Xerox PARC Technical Report CSL-89-1, January 1989.

For information on protocols see

Gursharan S. Sidhu, Richard F. Andrews, Alan B. Oppenheimer: *Inside AppleTalk*, Addison-Wesley, 1989.

Inside Macintosh, volume II, chapter 10, "The AppleTalk Manager," Addison-Wesley, 1985.

Inside Macintosh, volume V, chapter 28, "The AppleTalk Manager," Addison-Wesley, 1988.

Macintosh Technical Note #132, AppleTalk Interface Update, March 1988.

Macintosh Technical Note #225, Using RegisterName, February 1989.

Macintosh Technical Note #250, AppleTalk Phase 2 on the Macintosh, December 1989.

Thanks to Our Technical Reviewers
Mary Boetcher, Michael Gough, Kerry Lynn •

APPLE II Q & A

Q

The Apple IIGS® GS/OS Reference, *page 43, alludes to "an enhanced ProDOS® 8 QUIT call, which contains a pathname to an application to be launched." However, I find no mention of this enhancement in the* ProDOS 8 Technical Reference. *How do I use this call?*

A

The enhanced ProDOS 8 QUIT call allows you to quit to another application if GS/OS or ProDOS 16 has been booted. The enhanced ProDOS 8 QUIT call requires either of the following four-count parameter blocks:

Standard

```
dc.b $00    ;quit type
            ;normal
dc.w $0000  ;null
dc.b $00    ;null
dc.w $0000  ;null
```

Extended

```
dc.b $EE    ;quit type
            ;enhanced
dc.w path   ;addr of
            ;launch
            ;pathname
dc.b $00    ;reserved
dc.w $0000  ;reserved

path str 'myprog.sys16'
```

GS/OS patches ProDOS 8 to get control on a QUIT and launches the next program if the quit type is $EE.

The code to do this is not part of ProDOS 8; it doesn't fit in the kernel, and it can't go in the "quit code" because program selectors swap that out. The enhanced Quit call, therefore, works only when GS/OS has been booted.

Q

What's the difference between Apple IIGS System Software versions 5.0.3 and 5.0.4?

A

Apple IIGS System 5.0.4 includes the following changes:

- TOOL.SETUP for System Software 3.2 in May 1987 changed QDStartUp to make the cursor image handle safe and has now been changed for ROM 03 as well.
- QuickDraw Auxiliary no longer returns bogus errors for SeedFill and CalcMask in pure 640 mode, and a low-level stack imbalance has been corrected.
- The ImageWriter® and ImageWriter LQ drivers now spool to the User Path if the system was booted over AppleShare®. A bug concerning memory allocation has been fixed, and the drivers now check errors more robustly.
- The SCSI Manager no longer resets the SCSI bus when the Manager is started.
- The AppleShare FST now saves and restores the correct QuickDraw direct page locations when shielding the cursor to draw the AppleShare arrows.

Kudos to our readers who care enough to ask us terrific and well thought out questions. The answers to these puzzles have been supplied courtesy of our teams of technical gurus; our thanks to all. Special thanks to Matt Deatherage, C. K. Haun, Jim Luther, Eric Soldan, Dan Strnad, and Tim Swihart for the material in this Q & A column. •

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

APPLE II Q & A

Q

What do I need to get started with MPW IIGS?

A

MPW IIGS is a set of tools and languages that creates Apple II and Apple IIGS programs and object code under the Macintosh MPW development environment. The system requirements for MPW are detailed in *APDAlog*® in the MPW product description. In addition to MPW and a system suitable for it, you need the MPW IIGS Tools package, which contains necessary development tools like the linker and other useful tools such as the resource compiler, object module dumper, and ProDOS file duplicator. You will also need the MPW IIGS language of your choice—currently assembly, C, or Pascal.

Q

If a task in the Heartbeat Interrupt Task queue has not yet been executed (the tick counter has not yet reached zero), is it possible to store a zero into the **`TaskCnt`** *field to keep the system from ever executing the task?*

A

This will work fine. If you know where the count word is, then you can set it to zero to prevent yourself from being called. The system does not keep this information in a separate buffer; it checks the value in the queue header each pass through the Heartbeat queue, so if you were at 200 one pass, and then 0 the next, the system will not be bothered because it does not remember the previous value. And because the task is not executed unless the system itself decrements the count to zero, storing a zero into the **`TaskCnt`** field is a fine way to prevent a task from executing.

Q

Can run queue tasks remove themselves?

A

Yes, run queue tasks can call Desk Manager **`RemoveFromRunQ`** on themselves without difficulty.

Q

Can QuickDraw II Auxiliary's **`CopyPixels`** *call scale pixel images beyond* **`maxWidth`**?

A

No, but you can use the QuickDraw II **`SetBufDims`** call to increase the size of the QuickDraw buffer to beyond what was specified for the **`maxWidth`** variable in the **`QDStartUp`** routine.

Q

How do I port my Macintosh HyperCard® stack to run with HyperCard IIGS?

A

You can use HyperMover™, which is available on AppleLink® on developer

CDs. HyperMover allows HyperCard 1.2.5 stacks from the Macintosh to run with little or no modification on the Apple IIGS with HyperCard IIGS. HyperMover consists of two stacks, one for the Macintosh and one for the Apple IIGS. HyperMover for the Macintosh creates a folder containing files that describe the stack you wish to convert to the IIGS. This folder and the files it contains are then transferred to the IIGS via Apple File Exchange or an AppleTalk network. HyperMover for the IIGS then rebuilds a stack as close as possible to the original stack using the files contained in this folder. The most noticeable difference between the original and the rebuilt stack will be in the graphics. Because the IIGS and the Macintosh have such different sized screen displays, the graphics and objects of the rebuilt stack must be scaled to fit the IIGS screen, resulting in some loss of detail.

HyperMover contains several features designed to make the rebuilt stack as useful and as close to the original stack as possible. It can create scaled representations of Macintosh pictures, convert Macintosh sounds to IIGS sounds and Macintosh icons to IIGS icons, and transfer all HyperCard objects including backgrounds, cards, buttons, and fields and their attributes. However, because HyperMover is a stack, it cannot convert XCMD/XFCNs and cannot fix scripts that need specific Macintosh screen coordinates to function.

Q

Is HyperTalk the same in HyperCard IIGS as in Macintosh HyperCard?

A

Generally, HyperTalk® on the Apple IIGS is the same as HyperCard 1.2.5 HyperTalk on the Macintosh, but the HyperTalk on the Apple IIGS has an extended command set to support the features available in the Apple IIGS environment. New commands are included for setting color properties of objects, painting properties, and printing. A new property for buttons called the family property also has been added.

Q

Does HyperCard IIGS provide for extending the HyperTalk language?

A

Yes. External commands and functions, which are usually referred to as XCMDs and XFCNs, or externals as a general group, are functionally identical in the Macintosh HyperTalk and Apple IIGS HyperTalk software environments. XCMDs and XFCNs provide for extensions to the existing HyperTalk language and are called using the same methods as those for Macintosh HyperTalk. Modifications have to be made, however, to move existing source code for Macintosh externals into the Apple IIGS environment. HyperTalk callback procedures and interfaces for the Apple IIGS differ slightly from Macintosh HyperTalk.

MACINTOSH Q & A

Q

How can I determine the size of my application's memory partition?

A

It's really difficult to find the exact size of the memory partition the application is running under. Because there is little that an application can do to change its partition size (except to change the SIZE resource and then force a relaunch), the real concern would be to find the size of the available stack and heap. Included in the application's partition are the application parameters, jump table, application globals, and QuickDraw globals, the size of which is not easily determined. The only portions of an application's memory use that are adjustable at run time are the stack and the heap.

The stack and heap sizes are fixed within the boundaries of the entire application partition. Increasing one decreases the other. There are Memory Manager calls to change the size of the heap. To increase the stack size, you decrease the heap size. If you need to change the size of your stack or heap, be sure to do so at the start of your app, before memory at the end of the heap is allocated—before calling **`MaxApplZone`**, for sure.

Q

Is it still ok to adjust BufPtr?

A

The Macintosh documentation of BufPtr recommendations is based on the Macintosh 128 with 64K ROMs. MemTop is where the highest addressable physical RAM is located. The Macintosh IIci may store memory used by the video somewhere below this. The IIci does not have contiguous memory, and it is not safe to assume that you can adjust BufPtr based on the size of installed RAM. You may be running into the video RAM area used by the built-in video port.

Another problem will be that MemTop is different when running under MultiFinder. Since System 7.0 always runs MultiFinder, MemTop will be the top of the application's partition and not the physical RAM. In addition, while running Virtual Memory, MemTop may be a very large number such as 16 MB. We no longer recommend adjusting BufPtr to use the memory in this area.

Our current recommendation is to use the system heap. INITs can use the 'sysz' resource to increase the size of the system heap. If you are using memory at interrupt time, it should be in the system heap for virtual memory (VM) compatibility or held in real RAM with VM's HoldMemory call. Memory above BufPtr will be paged to disk while running VM.

Kudos to our readers who care enough to ask us terrific and well-thought-out questions. The answers to these puzzles have been supplied courtesy of our teams of technical gurus; our thanks to all. Special thanks to Pete "Luke Skywalker" Alexander, Mark Baumwell, Jeremy Bornstein, Rich Collyer, Dennis Hescox, Jim Luther, Guillermo Ortiz, Jim Reekes, Bryan Stearns, Robert Stobel, Forrest Tanaka, Vince Tapia, Jon Zap, and Scott "Zz" Zimmerman for the material in this Q & A column. •

Q

How do penguins know when it's safe to go in the water?

A

Penguins in the Antarctic face many perils, not the least of which are several species of sea mammals waiting at the edge of icebergs to nibble on their little penguin bods when they go in the water. So, penguins will just wander around on the edge of an ice floe until one of them accidentally falls in the ocean. When that happens all the penguins will stand by and watch to see if the fallen penguin gets eaten. If the penguin comes to harm it's back to business as usual on the iceberg. If the penguin is safe (apparently for a period of time known only to the other penguins) then the entire dissimulation leaps in, hunting for food.

Q

Is the maximum size of a resource still 32K?

A

No. There used to be a bug in the 64K ROMs that didn't allow you to write even multiples of 32K (that is, 32K-64K, 128K-192K). This was fixed in 128K ROMs. As of 128K ROMs, the resource size is limited to "maxlongint" bytes.

Q

How do I write a background-only application for MultiFinder?

A

A background-only application is similar to a standard MultiFinder-aware application except that it performs a task in the background with no user interface such as windows or menus. A background-only application must not initialize any Toolbox managers; for example, it must not call InitWindows or InitMenus. If a dialog needs to be displayed by a background application, the Notification Manager should be used.

Q

I am writing a Chooser PACK resource. When I change the flags that control which messages I will be sent, nothing seems to change. Why?

A

Chooser caches the flags from your Chooser PACK the first time it sees the file. From then on, changes are ignored until the PACK's name is changed, or until the Chooser's cache is flushed. To flush the cache, simply hold down the Command and Option keys when selecting Chooser from the Apple menu. When the cache is flushed, the system will beep.

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

MACINTOSH Q & A

Q

What can I do and not do while executing in interrupt code?

A

Interrupt code is most commonly used for I/O completion routines, VBLs, Time Manager tasks, and deferred tasks. Contrary to popular belief, the Deferred Task Manager will run your task at interrupt level. All code that runs at interrupt level must follow very strict rules. The most important rule is that the code cannot allocate, move, or purge memory, reference memory manager structures (that is, HUnlock), or call a Toolbox routine that would. This eliminates nearly, if not all, QuickDraw operations. For a more complete list of these Toolbox routines, refer to the *Inside Macintosh X-Ref*.

Additionally, interrupt code should avoid accessing a low-memory global or calling a trap that would access one. While MultiFinder is running, the applications' low-memory globals are being swapped in and out. Because of this, the interrupt code cannot rely on which application's globals are currently available. Even if **`CurApName`** is correct, the interrupt routine may be called while MultiFinder is in the process of swapping the applications' globals. This restriction is difficult to deal with because there is no documention as to which low-memory globals are swapped by MultiFinder, nor which globals are accessed by traps.

A typical example of this problem is interrupt routines that attempt to restore A5 by examining CurrentA5. This low-memory global is valid only while the current application is running at non-interrupt time. Thus, the Macintosh Programmer's Workshop (MPW) routine **`SetCurrentA5`** (or the obsolete **`SetupA5`**) cannot be used at interrupt level. It is necessary to place the application's A5 value somewhere it can be located while in the interrupt routine. This is documented in Technical Notes #180 and #208. In fact, the exact code you need is in #180.

Also, there are interrupt limitations while System 7 is running under Virtual Memory. Therefore, it is best to avoid interrupt code if at all possible. Move the functionality of the interrupt code into the application. For example, if you do require a VBL, limit the code to an absolute minimum. Also, set a global flag for the application to check in its event loop.

Q

Is the maximum size for global and local data still 32K?

A

Starting with MPW 3.0, the maximum size for global data in MPW became larger than 32K, using compiler and linker options -m and -srt. Local data is tougher, because local data is allocated by using the LINK instruction, for example, LINK A6,#$-380. With this relative addressing mode, you're constrained to the negative side of a 16-bit value for local space, which translates to 32K. That is, this limit is basically due to the Motorola processor architecture.

If you are allocating more than 32K either globally or locally, you might want to rearchitect your system to use dynamic (and theoretically unbounded, especially on virtual architectures) storage space.

Q

Is it possible to get PAL (Phased Alternate Lines) timing from the new 4•8 and 8•24 cards?

A

The 4•8 and 8•24 cards currently support NTSC (National Television Standards Committee) output and will support PAL output as well when the next ROM revision is put into production early this year. (The 8•24 GC card currently supports PAL.)

Q

What is a System Error 29? It's not in any of the documentation I have available. One of my applications has been reported to crash with this error occasionally.

A

When the Package Manager can't load in a PACK resource for any reason, it calls SysErr. There is a range of system error codes reserved for the situation in which the PACK resource couldn't be loaded. That range is 17 through 24. So if it couldn't load the List Manager package, which is PACK 0, the Package Manager adds 0 to 17 and calls SysErr with a code of 17. If it couldn't load the Standard File package, which is PACK 3, the Package Manager adds 3 to 17 and calls SysErr with a code of 20.

The problem is, we overflowed our PACK SysErr range. The Color Picker is PACK 12. If the Package Manager couldn't load in PACK 12, it adds 12 to 17 and calls SysErr with a code of … 29. And there you have it.

PACKs are loaded into the system heap. If there's not enough room in the system heap for a PACK, the system heap is expanded and the PACK is loaded in. If you set your application to take over the entire application memory space, the system heap can't grow anymore. The GetResource('PACK',12) call that the Package Manager makes fails: It adds 12 to 17, and calls SysErr with a code of 29. This could be what's causing the crashes in your case.

Q

How can we increase the stack space allocated by the DA Handler?

A

There is no simple, supported way to increase the stack space available to a DA. DAs were designed to provide easy access to simple tools from the user's desktop. As such, the environment of a DA is relatively limited. Now, with MultiFinder and especially with System 7.0, applications can provide all the functionality of a DA with none of the limitations. Under MultiFinder the DA Handler actually reduces the stack space available to a DA by about 25 percent. This limitation is not a problem for applications. If you need stack space beyond the bounds of the DA's environment, we encourage you to convert your DA to a full application.

MACINTOSH Q & A

Q

Does 32-Bit QuickDraw support packed PICTs? What's the technique for saving packed PICT formats? What compression schemes are supported?

A

Color QuickDraw has always supported packed PICTs. See *Inside Macintosh*, volume V, for details on how CLUT PixMaps are packed. Under 32-Bit QuickDraw, to pack direct RGB PixMaps in PICTs, call CopyBits with the packType field in the source PixMap set to one of the following constants that apply to direct RGB PixMaps:

0 default packing (pixelSize 16 defaults to packType 3 and pixelSize 32 defaults to packType 4)
1 no packing
2 remove pad byte (32-bit pixels only)
3 run-length encoding by pixel size chunks (16-bit pixels only)
4 run-length encoding, all of one component at the time, one scan line at a time (24-bit pixels only)

Scheme 4 will store the alpha channel also if cmpCount is set to to four. PackSize is not used and should be set to zero for compatibility reasons. See Inside Macintosh, Volume 6, for complete details.

Q

Is there any way to stop the Dialog Manager from playing with the `txSize` *and* `txFace` *fields of a dialog's* `grafPort` *so that I can draw Geneva 9-point text from within a userItem proc?*

A

Unfortunately, because the Dialog Manager forgets about your previous calls to TextFont and TextSize when you put up your dialog again, you will need to call TextFont and TextSize every time your userItem proc is called.

Q

When should the Color Manager be used and when should the Palette Manager be used?

A

The Palette Manager is by far the friendlier and more versatile of the two. It provides all the functionality you need to customize and animate the colors in your application. You shouldn't ever need to use the Color Manager unless you require custom color search and complement functions. Unless you really understand the Color Manager in detail, you are likely to have problems getting the Color Manager to work in a clean fashion.

When using the Palette Manager, applications following the rules will maintain their respective color environments safely as windows move back and forth from foreground to background, and from one screen to another. Accomplishing this with the Color Manager calls is not worth the effort. For additional information, see the Palette Manager article in this issue.

Q

I would like to make my fills print better. Currently, they come out as 72 dpi patterns. Is there a way that I can make them print at a higher resolution, but have my patterns still print as patterns?

A

To make your patterns print at the printer's resolution, you need to use Printing Manager PrGeneral's **`GetRslData`** and **`SetRsl`** opcodes to get and set the resolution, and you must scale the pattern to match. Let me explain.

If we do not scale our patterns up to the printer's resolution before print time, we would get "big chunky" patterns because the printer driver would need to scale the patterns on the fly from 72 dpi to its resolution. Therefore, we use the "cookie cutter" approach to "place" the pattern into the object that is being filled. The size of the "cookie cutter" (the destination Rect) depends on the "scaleFactor." For example, a "scaleFactor" of 2 will have a destination rect of 16 x 16. We will then CopyBits the pattern one square at a time into the object that is being filled.

You might find the article "Meet PrGeneral" from the July 1990 issue of develop useful for describing the functionality of PrGeneral in greater detail.

Q

What versions of Apple Color Printer software work with what system software versions? Can Apple Color Printer software distributed with System 6.0.7 work with System 6.0.5 and earlier versions? I am distributing LaserWriter® drivers with modified pgsz resources. I would like to cut back on the number of files I need to distribute.

A

The software sent with version 6.0.7 will work with all other 6.0.x systems. This should be the rule with most other LaserWriter printer software as well. Color has been supported since LaserWriter driver 6.0, for color depths of 8 bits or less. Depths greater than 8 bits must be converted before printing.

There really isn't any simple way to match up the version of released printer software with what version of the system it is compatible. LaserWriter 7.x is compatible with both System 7 and System 6. It's still prerelease software, however. Do not ship the preliminary LaserWriter 7.x driver with your application. You'll be able to ship the final LaserWriter 7 driver with your product as soon as System 7 is final.

Q

How can I use SndPlay to function asynchronously? It seems to ignore the async parameter.

A

To use SndPlay asynchronously, you must have allocated a sound channel without passing NIL as the chan parameter. There is one thing to be aware of in doing this, which often confuses developers. If the 'snd' resource being used with SndPlay specifies a 'snth' resource, then you cannot create the sound channel with a synth. Because of a Sound Manager bug that has been present in all releases through System 6.0.7, SndPlay has not worked correctly for a 'snd' resource specifying a 'snth', with a user channel initialized with a 'snth'. For example, the following code will fail:

```
SndNewChannel(myChan,
   sampledSound, init,
   @myCallBack);

SndPlay(myChan, sndHdl,
   async); {sndHdle is a
   sampled sound}
```

The Sound Manager attempts to link this 'snth' to the channel with every call to SndPlay. If the synthesizer has already been installed, the Sound Manager attempts to install it again, only this time as a modifier. The same 'snth' code ends up being installed more than once in the channel. If the 'snd' contains 'snth' information, then SndPlay can be used once and only once on a channel. A format 2 'snd' resource is assumed to be a sampled sound. For format 1, check the number of snths specified in the 'snd' and then check each one. The latest version of SoundApp has source code that does these tests.

This limitation has been fixed in System 7. In System 7, SndPlay can be called any number of times on a channel. For older system releases you need to create and dispose of the channel each time after calling SndPlay, as in the following code:

```
#include   <Resources.h>
#include   <Sound.h>

#define    TRUE    0xFF
#define    FALSE   0

main()
{
   Handle         Sound;
   SndChannelPtr  chan;
   int            i;
   OSErr          err;

   Sound = GetResource
   ('snd', 100);
   if (ResError() != noErr
      || Sound == nil)
   Debugger();

   for (i = 0; i < 3; ++i)
   {
      chan = nil;
      err = SndNewChannel(
         &chan, 0, 0, nil);
      if (err != noErr)
         Debugger();

      err = SndPlay (chan,
         Sound, FALSE);
      if (err != noErr)
         Debugger();
```

```
        err =
            SndDisposeChannel(
            chan, FALSE);
        if (err != noErr)
            Debugger();
    }
}
```

A good method for playing sampled sound asynchronously on any Macintosh is to create a sound channel and use the bufferCmd. Find the sound header from the 'snd ' resource and pass the pointer to this in the bufferCmd. Use this with SndDoCommand or SndDoImmediate. To determine when the sound has completed so that you can know when to dispose of the channel, send a bufferCmd with SndDoCommand (in order to queue it) after the bufferCmd. Once your callback procedure is called, set a global signalling that the sound has finished. The new Sound Manager (System 6.0.7 and beyond) supports a new call, SndChannelStatus, which will tell you if the channel is playing a sound or not. Instead of the callback procedure, you can poll the channel's status to determine when to dispose of the channel. Example code using the callback procedure can be found in the DTS sample code SoundApp.

Q

What is the difference between North and West?

A

North is an absolute direction on the globe. Once you are on the North Pole it is impossible to go any "further North." West, on the other hand, is a relative position. No matter where you are on the surface of the Earth, it's always possible to go further West.

Q

I have found that sounds recorded at good or better quality will not play with system software prior to 6.0.7. That's expected, but the fact that SndPlay does not return an error message is not. How can I check to see if a sound is compressed when running older system software?

A

There is a byte in the SoundHeader data structure (and thus in a 'snd' resource), called "encode." If the sound is compressed, the value of this byte will be $FE, which is defined as the constant cmpSH in the headers for the Sound Manager.

Q

Why do golf balls have pocks?

A

As counter intuitive as it may seem at first, the pocks on golf balls actually make them fly farther. A golf ball builds turbulence in front of it as it flies through the air. If the ball is pocked, the turbulence "fills" the pockets as the ball spins, resulting in less resistance and more flight. 747s have pocks on their wings for the same reason. Dolphins presumably have these pocks on their fins for this purpose as well, but to date none of them have come out publicly and said so.

MACINTOSH Q & A

Q

We generate sounds using the Sound Driver's four-tone synthesizer. Our application must run on all Macintosh computers and all system software versions starting with System 6.0. According to an early version of Inside Macintosh, *Volume VI, the new Sound Manager's wave-table synthesizer, which replaces the Sound Driver's four-tone synthesizer, does not perform as expected on some Macintosh systems. When should we use the Sound Manager and when should we use the Sound Driver?*

A

The Sound Driver is no longer supported, as of System 6.0.7. The wave-form synthesizer in the Sound Manager released with 6.0.7 does not work correctly for non-Apple Sound Chip machines (Macintosh Plus, SE, Classic, and LC), but this will be fixed in System 7. If you need to use the wave-form synthesizer for non-ASC machines running 6.0.7, you could try the Sound Driver with 6.0.7 on the chance it'll work for your purpose. Use the Sound Driver for non-ASC machines running System 6.0.5 and earlier.

Q

Does stereo work? I was hoping to init the left and right channels separately (using **`initChanLeft`** *and* **`initChanRight`***) and send different sampled sounds out both channels. But the Sound Manager documentation says stereo is not supported. I figured this would be a "cheap" way of playing two sounds at the same time, sending them out the left and right channels and letting the Macintosh mix them together (or telling the user to flip the MONO switch on their stereo).*

A

Stereo and mixing multiple channels are new features of the Sound Manager released with System 6.0.7. If you create a mono channel, the sounds come out both speakers. If you create a left channel, it is a mono sound coming from only the left channel. Alternatively, creating a right channel will only come from the right. If you create a stereo channel, then the sound's left or right position is determined by the sound header you use (with the bufferCmd). A stereo sound is only supported by a compressed or extended sound header. You cannot control the left or right panning of a stereo sound. This is only determined by the sound header and its interleaved data. The left or right init params will have no affect on a stereo channel. True stereo sounds can only occur by using a stereo sound header. Two mono channels, one for the left speaker and one for the right, could be opened for the affect that most developers want.

Only the Macintosh SE/30 and the Macintosh IIsi have both the left and right sources mixed to the internal speaker. A stereo sound on all other Macintosh systems have the left source only sent to the internal speaker. The right source is only sent to the external port, and it isn't possible to determine if the external port is in use or not.

Q

Is it necessary to lock a 'snd' resource that is to be played asynchronously with SndPlay?

A

Yes, if you are playing a sound resource asynchronously with SndPlay, then you have to lock the sound. SndPlay will restore the handle's state as soon as the trap returns to the caller. If the call is asynchronous, the handle's state is restored immediately after calling SndPlay, before the sound finishes playing.

Q

When converting stacks from HyperCard 1.2.5 to 2.0, the default "fixed line height" setting for text fields sometimes enlarges the space required by text and destroys the layout of the screen. For example, text tends to disappear outside the edges of the field, or be misaligned. Deselecting "fixed line height" corrects the problem in most instances, but there can still be a slight discrepancy in the amount of space taken up by identical fonts in identical fields between 1.2.5 and 2.0, such that layouts are disrupted even if "fixed line height" is not selected.

A

Inherent in the design of HyperCard 2.0, the way a field is displayed may be different in HyperCard 2.0 than in HyperCard 1.2.5. Converting a stack to HyperCard 2.0 from HyperCard 1.2.5 may require that fields be tweaked to appear properly. Therefore, developers may want to provide special versions of their stacks for use with HyperCard 2.0, regardless of whether features specific to HyperCard 2.0 are incorporated.

Q

Since I have received HyperCard 2.0 I have converted several stacks for 1.x to 2.0 and have experienced several problems with scripts that worked fine in 1.x. I have recently learned that HyperCard 1.x will NOT run on System 6.0.7 and later versions. For HyperCard 1.x users this results in stacks that cannot be used in HyperCard 2.0. For HyperCard stack developers this presents a nightmare in converting old 1.x stacks for users into 2.0 stacks. Since encountering these problems I have changed my scripts in 1.x so that when converted they work. What is Apple's position on "seamless conversion" of HyperCard 1.x stacks to HyperCard 2.0?

A

Most functioning 1.x scripts will work without modification under HyperCard 2.0. However, HyperCard 2.0 is a bit more strict in enforcing some of the grammar of HyperTalk. For example, keywords can no longer be used as variable names under 2.0. Under 1.x, keywords could be used as variable names, but the documentation specifically warned against doing this.

SYSTEM 7.0 Q & A

Q

When I use the name and vRefNum returned from FindFolder, I always get a fnfErr from OpenRFPerm. Why?

A

FindFolder returns both a vRefNum and a DirID. They both must be used to identify the folder. Instead of using OpenRFPerm,which takes only a vRefNum, try HOpenResFile. Avoid using PBHSetVol! (See Technical Note #140 for more information.)

Q

We want to use OpenCPicture for higher resolution, not for color per se. Can OpenCPicture in System 7.0 be used with non-Color as well as Color QuickDraw Macintosh computers?

A

Yes, with System 7.0, OpenCPicture can be used to create extended PICT2 files from all Macintosh computers. Under System 6.0.7 or later, you must test for 32-Bit QuickDraw before using OpenCPicture. You can do this by calling Gestalt with the gestaltQuickDrawVersion selector. If it returns gestalt32BitQD or greater, then 32-Bit QuickDraw is installed.

Q

Can the Communications Toolbox be used in a DA? InitCM has to be called and the manual says it should be called only once.

A

Yes, it is all right to call CTB Initialization routines from CODE resources. This includes a DRVR, cdev, or INIT.

Q

Can I use my 8•24 GC card with System 7.0ßx? I was told the GC misbehaves with any and all nonlinear address mapping, such as System 7.0 and A/UX. Is this true? Can this be fixed with CODE/INIT patch? Is it scheduled to be fixed in the near future?

A

The 8•24 GC card software version 1.0 is not intended to run in any environment involving virtual memory management. A VM-compatible version of the GC software will be made available when System 7.0 is final. Until then, the card can be used as a video buffer, and System 7.0 preliminary software might run with VM switched off.

Kudos to our readers who care enough to ask us terrific and well-thought-out questions. The answers to these puzzles have been supplied courtesy of our teams of technical gurus; our thanks to all. Special thanks to Pete "Luke Skywalker" Alexander, Jim "Im" Beninghaus, Rich Collyer, Guillermo Ortiz, Forrest Tanaka, and Scott "Zz" Zimmerman for the material in this Q & A column. •

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

Q

Under what System 7.0 and System 6.0 conditions is it legal to call the QDError function?

A

Under System 7.0, QDError can be called from all Macintosh computers. (System 7.0 supports RGBForeColor, RGBBackColor, GetForeColor, and GetBackColor for all Macintosh computers as well.) On a non-Color QuickDraw Macintosh, QDError always returns a "no error." Under System 6.0, QDError cannot be used for non-Color QuickDraw Macintosh systems.

Q

Why do some CopyBits transfer modes produce different results for System 7.0 than for System 6.0?

A

Under System 6.0, the srcOr, srcXor, srcBic, notSrcCopy, notSrcOr, notSrcXor, and notSrcBic transfer modes do not produce the same effect for a 16- or 32-bit (direct) pixel map as for an 8-bit or shallower (indexed) pixel map. With Color QuickDraw these classic transfer modes on direct pixel maps aren't color based; they're pixel value based. Color QuickDraw performs logical operations corresponding to the transfer mode on the source and destination pixel values to get the resulting pixel value.

For example, say that a multicolored source is being copied onto a black and white destination using the srcOr transfer mode, and both the source and destination are 8 bits per pixel. Except in unusual cases, the pixel value for black on an indexed pixel map has all its bits set, so an 8-bit black pixel has a pixel value of $FF. Similarly, the pixel value for white has all its bits clear, so an 8-bit white pixel has a pixel value of $00. CopyBits takes each pixel value of the source and performs a logical OR with the corresponding pixel value of the destination. Using OR to combine any value with 0 results in the original value, so any pixel value ORed with the pixel value for white results in the original pixel value. Using OR to combine any value with 1 results in 1, so any pixel value ORed with the pixel value for black results in the pixel value for black. The resulting image shows the original image in all areas where the destination image was white and shows black in all areas where the destination image was black.

Take the same example, but this time make the source and destination 32 bits per pixel. The direct-color pixel value for black is $00000000 and the direct-color pixel value for white is $00FFFFFF. CopyBits still performs a logical OR on the source and destination pixel values, but notice what happens in this case. Using OR to combine any source pixel value with the pixel value for white results in white, and using OR to combine any source pixel value with the pixel value for black results in the original color.

SYSTEM 7.0

Q & A

The resulting image shows the original image in all areas where the destination image was black and shows white in all areas where the destination image was white—roughly the opposite of what you see on an indexed pixel map.

The newer transfer modes addOver, addPin, subOver, subPin, adMax, and adMin work consistently at all pixel depths, and often, though not always, correspond to the theoretical effect of the old transfer modes. For example, the adMin mode works similarly to the srcOr mode on both direct and indexed pixel maps. Also, 1-bit deep source pixel maps work consistently and predictably regardless of the pixel depth of the destination even with the old transfer modes.

Under System Software 7.0, the old transfer modes now perform by calculating with colors rather than pixel values. You'll find that transfer modes like srcOr and srcBic work much more consistently even on direct pixel maps.

Q

Is the Macintosh printing architecture different for System 7.0?

A

No changes were made to the printing architecture for System 7.0. Printer drivers were revised for System 7.0 to support TrueType, to be completely 32-bit clean, and to fix bugs, but the printing architecture remains the same for System 7.0.

Q

BitMapToRegion does not work as described in Technical Note #275 for a PixMap with baseAddr = (NuBus address). Which calls support PixMap 32-bit base addressing with pmVersion = 4?

A

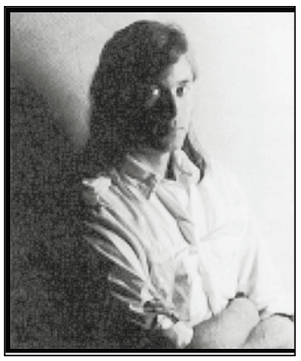
As of System 7.0ß1, BitMapToRgn cannot handle a bitmap whose base address is in the NuBus™ address space or any bitmap that requires 32-bit addressing. The problem will be fixed for System 7.0's final release. As far as we know, BitMapToRgn is the only call that doesn't yet support 32-bit addressed bitmaps.

Q

Can the LaserWriter 7.x driver be used with System 6.0?

A

Yes! LaserWriter 7.x is compatible with both System 7.0 and System 6.0. It's still prerelease software, however. Do not ship the preliminary LaserWriter 7.x driver with your application. You'll be able to ship the final LaserWriter 7 driver with your product as soon as System 7.0 is final. To use the new LaserWriter driver with an AppleShare print spooler, you need a special LaserPrep, available on developer CDs and on AppleLink.



THE VETERAN NEOPHYTE

LISP, COLOR ICONS, AND LAYERS

DAVE JOHNSON

I recently started learning Lisp, in order to write a color icon editor for a project in ACOT℠ (that's Apple Classroom of Tomorrow). The people behind ACOT are creating an environment in which kids can build and explore simulated ecologies, and I wanted to help, but didn't know Lisp well enough to be of much use. A color icon editor was needed, and it seemed like a straightforward and painless little project to cut my Lisp teeth on. Hah.

I did learn lots of Lisp, but I spent a disproportionate amount of time learning the low-level system interface in Macintosh Allegro Common Lisp (MCL) and wrestling Color QuickDraw to the ground. For you CopyBits fans, did you know that CopyBits always assumes that the destination PixMap is on the current GDevice? I guess I already knew this—it's documented all over the place—but the implications never affected me before. I needed to convert a 4-bit PixMap to 8 bits, so hey, let's use CopyBits, right? Wrong. The colors got munged every time, because color mapping kicked in and the source's color table didn't match the GDevice's. For the gory details, see Technical Note #277, especially the section on color mapping. If you like CopyBits, you'll *love* this tech note.

I was curious what others thought of Lisp, so I asked around a little. Here are some of the responses I got:

Functional languages are cool if you have good libraries, but all those parentheses are a pain in the ass.
—Bryan "Beaker" Ressler, C programmer.

It's the shortest distance between conception and realization.
—Matthew MacLaurin, self-admitted Lisp junkie.

I hate it.
—Neil Day, who was forced to write the Tower of Hanoi iteratively in an introductory Lisp course.

It's a great productivity tool, and Common Lisp provides a rich (though perhaps Byzantine) programming environment.
—Gregg Williams, technical writer and sometime Lisper.

(expectant look)
—Natty, my dog.

There are several immediately apparent things about Lisp that are foreign to people used to C or Pascal. Data typing is nonexistent unless you want it: Any variable can hold any type of data at any time. Functions can be data, too, and can be passed around all over the place. Everything is cozily wrapped in parentheses many levels deep (after a while, this is somehow comforting). Context is all important and omnipresent. Changes can be immediately tried out, so for prototyping (and for those of us who thrive on immediate gratification) it's a joy to work in. For those lacking in discipline, Lisp can help to create a real mess (of course, any language can do that for you, it's just easier in Lisp). Because it is so forgiving, it encourages my own built-in "middle-out" design

strategy, which in the long run isn't terribly efficient, although lots of fun. With a little self-control, of course, this problem would go away (left as an exercise for the reader).

Writing Lisp code that is QuickDraw intensive is kind of a pain at first. MCL provides great libraries for basic QuickDraw tasks, but if you want to get fancy, you have to do it yourself. For the icon editor, I needed lots of little utility routines to do stuff like find a particular color's pixel value in the color table, add a color to a color table, copy a cicn, change the depth of a cicn, build new cicns from scratch. Nearly half my code consists of these little utilities. They'd be needed regardless of the language, I guess, but writing them in Lisp required me to learn the low-level system interface much more thoroughly than I originally intended. This made me grumble a little, but after I'd gotten over the initial syntactical hump, low-level access became transparent and largely effortless.

The other half of the code was much Lispier. I used the built-in object system (Object Lisp, since I was using MCL 1.3.2. Now, in MCL version 2.0, it's the Common Lisp Object System, or CLOS), and I found that it successfully isolated me from most of the grungy system details like events and window handling (that's what it's supposed to do, right?). I haven't done a lot of object programming, so I can't make incisive comparisons with other object systems, but I liked it.

The sort of layered, threaded structure of Lisp, and the continual "level switching" I had to do during development, got me thinking about how the machine is getting progressively more distant from the software I write. It seems that I write software to live on top of other software, not software to live on a machine. Object programming is a kind of layer creation, in that a well-designed object hides lots of details from the user of the object. MacApp is a layer (a thick one), HyperCard is also a layer (a *really* thick one), the Mac® Toolbox is a layer, and so on.

More and more, programmers need to be comfortable stuffed between these layers. Here's the hard truth: YOU NO LONGER HAVE TOTAL CONTROL OF THE MACHINE. Once upon a time, in the dim and distant past, programmers had absolute power over every aspect of the computer. There wasn't even any such thing as a user! A programmer was God in a monotheistic universe. Now there are all sorts of software smoke screens between your code and the machine itself. You are no longer God; at best, you are a minor demigod in charge of shoes, or something. A long time ago I read a discussion of Macintosh programming, and one person compared it to sitting in a dark closet by yourself, and occasionally answering a note that is passed under the door from the outside. I think that's exaggerating a little, but the point is valid. You no longer need to know everything that's going on in the house; you can just be responsible for your own room. At least, that's the idea . . .

One persistent problem is that you have to depend on the reliability of the other code. When my icon editor was almost done, I found a memory leak. Two tiny handles were left on the heap after closing the editor. It took me almost a week (and some expert help) to track it down to a bug in the MCL object system. Obviously, this diluted the benefits gained by using the system.

Overall, though, I really do think that this division of labor, this layering, is a good thing. It lets people find the niche they like best, and ignore much of the rest if they want to. Often it is an incredible time-saver to learn to use others' code rather than learning to do what they did from scratch. Ideally, the layers will insulate you completely from irrelevant detail, and allow you to focus on your task. We're not there yet, but someday, maybe, you can actually stop inventing the wheel.

DEVELOPER ESSENTIALS: VOLUME 2, ISSUE 1

Here's the latest Developer Essentials *disc. In addition to* develop *and related code, on this issue of the disc you'll find tools and information we think every developer should have. These pages highlight what's on the disc, but once you start browsing, you'll also find a few surprises.*

To use the disc, you need a CD-ROM drive and the appropriate cables and conectors. Refer to your CD-ROM drive's owner's manual for detailed information about connecting the drive to your particular machine.

For a Macintosh, you need at least 1 MB of memory, System 4.1 or later, and Finder 5.4 or later. In addition, you need to copy the Apple CD-ROM INIT that comes with the CD drive startup disks into your System Folder. For an Apple II, your SCSI card must have Rev C or later ROM. With ProDOS, no special setup is required. If you use GS/OS, you must use the Installer on System Disk 4.0 or later to install the CD-ROM driver on your startup volume.

develop

You've read the articles, you've bought the arguments, and now it's time to write your own code. The idea is that you don't have to waste your time typing the example programs—just mount this handy CD-ROM, then copy and paste. We've included develop as well as the code from each of the articles to help you avoid typos. So, browse around, take what you need, and save the rest for a rainy day. Each new issue of *Developer Essentials* will archive all of the back issues of the journal and the code. So look forward to one-stop searching coming soon to a CD-ROM near you.

ATG

The ATG folder contains a sampling of work being done by ATG researchers. ATG has projects for software and hardware, for artificial intelligence and education research, for human interface and library science. We've tried to bring you a sampling from all areas. We hope you'll find something useful, something interesting, and maybe even something amusing.

SpInside Macintosh

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

Apple II

Nine folders full of fantastic findings! Here you'll find a Programming and Utilities folder which contains such goodies as MPW IIGS Interfaces, and Apple II Getting Down to Basics. Check the Info Island folder to find out how to create icons that the AppleIIGS Finder can recognize. The Storage folder contains information on the 5.25 inch Disk Holder, CDSC Setup, SCSI Utilities, and SCSI Driver Shell, all for Apple II. If you need Apple II system software it's located in both the Apple II Systems folder and in the Apple.II.partition folder. You'll find HyperCard in the HyperCard IIGS folder. Be sure to browse through the Imaging, AppleShare and Universal Access folders for even more Apple II goodies.

DTS Technical Notes and Sample Code

Could you use a few programming tips and techniques? (Couldn't we all?!) All Apple II and Macintosh Technical Notes and Sample Code programs are here for your reference. Technical Notes are updates to existing technical documentation, useful hints and tips, and special coverage of technical topics. Included as well are Human Interface Notes which will help you develop uniform user interfaces in Apple II and Macintosh applications. In the Interim Toolbox Docs folder you'll find the World Wide Guide to System Software including release notes. This folder also contains the Sound Manager if you want your application to be able to create, modify, and play sounds. Other latest and greatest development information can be found in the Misc Tech Docs folder. One of these is Learning to Drive which is a detailed description of the Printing Manager: a guide to printing on the Macintosh. Finally, look in the Apple Publications folder to find an Apple Publications Style Guide and Glossary.

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*, Q & A Stack, and the Human Interface Notes Stack.

Macintosh Q & A Stack

Got a tough development question? Try the Q & A Stack, which is a collection of the most frequently asked questions DTS receives from developers. Organized by subject, this stack answers the questions within and includes cross-references to *SpInside Macintosh* and the Macintosh Technical Notes Stack.

International System Software

Developer Essentials includes all the latest international versions of Macintosh system software. In addition, look for KanjiTalk Toolkit, KanjiTalk 6.0 Docs, and Taiwan Chinese Font Option Kit. (You must have a Macintosh to run DiskCopy and create floppy disks from these images.) Explore!

International HyperCard

Need the latest version of HyperCard? Look no further. *Developer Essentials* includes the latest international versions of this "software erector set" in DiskCopy image format.

U.S. SystemSoftware/HyperCard

Here you will find the versions of system software from 0.1 to 6.0.5 which you can copy right to a floppy using DiskCopy. With access to these versions of system software, you can have compatibility with applications written under different versions. In addition, look for HyperCard U.S. versions 1.2.2, 1.2.5, and 2.0 all of which come complete with an idea stack. Try one! 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.

Essential Utilities

Do you need quick access to either TeachText 1.2 or Apple DiskCopy 4.1? Look here!

Now you know 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.

SCANNING FROM PRODOS

This article shows just how easy it is to include support for scanner hardware in your application program. With just a little effort, you can add significant functionality to your program.

MATT GULICK

In this article, we explore using the Apple Scanner (a flatbed scanner) and the Apple II High-Speed SCSI Card with either an Enhanced Apple IIe computer or an Apple IIGS computer running the ProDOS-8 operating system. (A future article will cover GS/OS.) The concepts presented here can be used for any scanner that can be connected to an Apple IIe or Apple IIGS via the Apple II SCSI card.

For this article, we limit our discussion to the graphics modes available on the Apple IIe (HiRes and Double HiRes modes). These modes are more limited in resolution and color generation than the Super HiRes mode available on the IIGS, but they allow our sample program to run on most of the current Apple II family of systems in use today. We focus on 1-bit-per-pixel halftone and line art images. In so doing, we are able to display the data on the screen easily.

PLAYING HIDE-AND-SEEK WITH THE SCANNER

...98, 99, 100. Ready or not, here we come. Under the ProDOS-8 operating system, we don't have access to the loaded drivers that have been written for the GS/OS environment. Since the scanner is a character device, data is returned in bytes rather than in blocks. ProDOS-8 can't help us read from character devices, so we need to walk the slots looking for the card we want and then talk to the card directly to find the device we want.

APPLE HIGH-SPEED SCSI CARD, WHERE ARE YOU?

We must first find which slot the high-speed SCSI card is in. We start at slot 7 and work our way down. In the following code segment, we look for a SmartPort device in the current slot. If one is found, we must determine if it is a SCSI card that supports extended SmartPort calls. Finally, we need to make sure that this is the type of card we want. In other words, "Is this card from a vendor whose command set I understand?" See Code Sample 1.

MATTHEW GULICK According to his business cards, Matt Gulick is an all-around SCSI (say it out loud) guy—who hates to shave and refuses to wear shoes except when meeting with someone with a title of VP or higher. He dearly loves the strict dress code, highly regimented working hours, and totally controlled environment at Apple. His career here was preordained by his being "genetically defective at birth."

This condition first visibly manifested itself at age 12 when he began reading computer punch cards for fun. He did temporarily buck his computer industry destiny by studying pre-vet medicine at Brigham Young University. However, after college he got back on track by working as an "electronic stuff" sales rep, and then he programmed for ParaMIS. Now he feels he's running the perfect scam: getting paid to play

```
;*****************************************************
;
;     CODE SAMPLE 1
;
;     In this first code segment, we walk the slots starting
;     at slot 7, looking first for a card of any kind. Once
;     found, we check the ID bytes for a SmartPort card.
;     Once found, we check the ID Type byte to see if it is
;     a SCSI card. If the card passes all these tests, we
;     then issue a Device $00 Status $00 call to further
;     ensure that this is the Apple II High-Speed SCSI Card.
;
;*****************************************************

find_card
                                ;
                                ; Save the current Zero
                                ; Page values before
                                ; using them.
                                ;
            lda   <My_ZPage
            pha
            lda   <My_ZPage+1
            pha
                                ;
                                ; Start at slot 7.
                                ;
            lda   #slot_7
            sta   <My_ZPage+1       ;Zero Page
            sta   slot+1            ;For Safe keeping
            stz   <My_ZPage
            stz   slot
                                ;
                                ; Is it a SmartPort card?
                                ;
@chk_smart  ldy   #Blk_sigl
            lda   (My_ZPage),y      ;Block_device Signature Byte
            cmp   #$20              ;#1 = $20
            bne   @next_slot

            ldy   #Blk_sig2
            lda   (My_ZPage),y      ;Block_device Signature Byte
            bne   @next_slot        ;#2 = $00
```


with computers. A member of the Dr. Demento Fan Club (DDFC), he fears his wife will sue him for dementing his four children, who only know the Weird Al versions of song lyrics. •

```
            ldy   #Blk_sig3
            lda   (My_ZPage),y      ;Block_device Signature Byte
            cmp   #$03              ;#3 = $03
            bne   @next_slot

            ldy   #SPort_sig
            lda   (My_ZPage),y      ;SmartPort Signature Byte
            bne   @next_slot        ;#1 = $00
                                    ;
                                    ; We have a SmartPort
                                    ; device. Is it SCSI with
                                    ; Extended SmartPort?
                                    ;
            ldy   #SPort_ID
            lda   (My_ZPage),y
            and   #Ext_SPort+\
            SCSI
            cmp   #Ext_SPort+\
            SCSI
            bne   @next_slot
                                    ;
                                    ; Is it an Apple II
                                    ; High-Speed SCSI Card?
                                    ;
            jsr   is_it_appl
            bcc   @exit
                                    ;
                                    ; Check the next slot.
                                    ;
@next_slot  lda   <My_ZPage+1
            dec   a
            sta   <My_ZPage+1
            sta   slot+1
            cmp   #slot_1
            bge   @chk_smart
            lda   #No_dev           ;No Device Error
                                    ;
                                    ; Clean exit
                                    ;
```

```
@exit       tax
            pla

            sta   <My_ZPage+1
            pla
            sta   <My_ZPage
            txa

            cmp   #$01              ;Set Carry if Non-Zero.
            rts
                                    ;
                                    ; This routine determines
                                    ; if the card is the new
                                    ; high-speed SCSI card.
                                    ;
is_it_appl  ldy   #$ff
            lda   (My_ZPage),y
            clc
            adc   #$03              ;Set SmartPort Entry Address.
            sta   card_ntry

            lda   <My_ZPage+1
            sta   card_ntry+1

            jsr   call_card
            dc.b  $00               ;Status Call Command Number
            dc.w  stat_list1
                                    ;
                                    ; Check the results.
                                    ;
            lda   stat_data+2       ;Low Byte of Vendor ID
            cmp   #$01              ;Must be $01
            bne   @non_apple

            lda   stat_data+3       ;High Byte of Vendor ID
            bne   @non_apple        ;Must be $00

            lda   stat_data+4       ;Low Byte of Version
            bne   @non_apple        ;Should be Null

            lda   stat_data+5       ;High Byte of Version
            bne   @non_apple        ;Should be Null
```

```
            clc                     ;Acc. 0 by previous LDA
            bra   @done
@non_apple  lda   #No_dev           ;Device not found

            sec
                                    ;
                                    ; Restore ZPage.
                                    ;
@done       pha
            php
            lda   slot
            sta   <My_ZPage
            lda   slot+1
            sta   <My_ZPage+1
            plp
            pla
            rts

slot        dc.w  $0000

;******************************************************

call_card   jmp   (card_ntry)

card_ntry   dc.w  $0000

;******************************************************

stat_list1  dc.b  $03               ;PCount = 3
            dc.b  $00               ;Device = Card
            dc.w  stat_data         ;Data returned here
            dc.b  $00               ;Get Host Status Call

;******************************************************

stat_data   dcb.b 64,0              ;Our Buffer

;******************************************************
```

THE SCANNING PROCESS

The scanning process involves five steps for your application, described briefly below. For general information about scanner technology and terminology, see the *Apple Scanner Reference*.

1. Initialize the scanner parameters
You must set the scanner parameters before you start a scan. These parameters determine how much space the image needs.

Use these commands:
MODE SENSE ($1A)
MODE SELECT ($15)
SEND ($2A)
DEFINE WINDOW PARAMETERS ($24)

2. Define an image buffer
The image buffer is free memory within the computer system that holds the bitmap image returned by the scanner. The size of the buffer dictates the amount of data you can retrieve from the scanner and thus the size of the image. If an image is larger than the available free memory, you can spool it to disk for later retrieval.

3. Start the scan
After you set the parameters, you can issue a scanner command to start scanning. When the scanner receives this command, it scans the image and places it in its internal memory.

Use this command: SCAN ($1B)

4. Request the scanned data
You must read the image from the scanner as it is placed in the scanner's internal memory. Because the scanner's memory can hold only a small portion of the image being scanned, and because you must read the data to allow the scan to continue, you should poll the scanner promptly.

Use these commands:
GET DATA STATUS ($34)
READ ($28)

5. Save the image to a file
You can save the data in a number of formats: HiRes and Double HiRes for the Apple II family, Super HiRes for the Apple IIGS, and PICT or any other Macintosh image format. You can also store the data in other formats, such as GIF. The choice is yours.

FINDING THE SCANNER IN A HAYSTACK

Now that we've found the card, or at least *a* card (there may be more than one), we need to ask the card, politely of course, if it has seen the scanner and if so, where. See Code Sample 2.

"Excuse me SCSI card, we're taking a census and would like to ask you a few questions if you don't mind. How many devices live at this slot? I see, and are any of them by chance character devices? Hmmm, too bad. I'll try the next slot. Sorry to bother you, and thank you for your time."

. . . a few slots later . . .

"Hi, we're taking a poll and would like your response to a few short questions. How many devices live at this slot? That many, great. Are any of them character devices? Getting warmer. May we come in to talk to them? Thank you."

```
;*****************************************************
;
;     CODE SAMPLE 2
;
;     In this code segment, we walk the unit numbers from the
;     SCSI card starting at unit 2 and going to unit 0 to
;     get the actual unit number count. Once this is
;     done, we start at unit 1 and walk forward until we
;     find the scanner.
;
;*****************************************************

      find_scanr
                                ;
                                ; First we issue a
                                ; Status call to device
                                ; number 2. This call
                                ; forces the card to
                                ; build its tables if it
                                ; has not yet done so.
                                ;
            lda   #$02
            sta   dev_num2
            stz   stat_code2

            jsr   call_card
            dc.b  $00           ;Status Call Command Number
            dc.w  stat_list2
            bcs   @error
                                ;
                                ; Now call unit 0 to
                                ; find out the total
                                ; device count.
                                ;
            stz   dev_num2
            jsr   call_card
            dc.b  $00           ;Status Call Command Number
            dc.w  stat_list2
            bcs   @error

            lda   stat_data2    ;Get the Total Device
            sta   dev_count     ;Count.
```

```
                lda   #$03              ;Set up for DIB Status
                sta   stat_code2        ;calls.

@loop           lda   dev_num2          ;First time we increment
                cmp   dev_count         ;a zero giving a device
                bge   @error            ;number of 1.

                inc   dev_num2
                jsr   call_card
                dc.b  $00               ;Status Call Command Number
                dc.w  stat_list2
                bcs   @error

                lda   d_type
                cmp   #$08              ;Is it Type = Scanner?
                bne   @loop             ;No

                lda   d_stype
                cmp   #$A0              ;Subtype = $A0?
                bne   @loop             ;No
                                        ;
                                        ; Scan string is a Pascal
                                        ; string (a length byte
                                        ; followed by ASCII). We
                                        ; want to make sure that
                                        ; both the length and the
                                        ; text in 'scan_str' match
                                        ; the data returned in
                                        ; 'id_str_len' and
'id_str'.
                                        ;
                ldx   id_str_len
@str_loop       lda   id_str_len,x
                cmp   scan_str,x
                bne   @loop
                dex
                bne   @str_loop

                lda   dev_num2          ;We have our scanner.
                sta   scan_dnum
                lda   #No_Err
                clc
                rts
```

```
@error      lda   #No_dev           ;Device not found.

            sec
            rts

;******************************************************

scan_str    dc.b  'APPLE    SCANNER ';4 Spaces between
                                     ;1 Space after
dev_count   dc.b  $00

;******************************************************

scan_dnum   dc.b  $00               ;Scanner Device Number

;******************************************************

stat_list2  dc.b  $03               ;PCount = 3
dev_num2    dc.b  $00               ;Device number
            dc.w  stat_data         ;Data returned here
stat_code2  dc.b  $00               ;Status Code

;******************************************************

stat_data2                          ;Our Buffer. Used over.
d_stat      dc.b  $00               ;Device Status Byte
blk_low     dc.b  $00               ;Block Count (Low)
blk_mid     dc.b  $00               ;Block Count (Mid)
blk_hi      dc.b  $00               ;Block Count (High)
id_str_len  dc.b  $00               ;ID String Length
id_str      dcb.b 16,$00            ;ID String (16 Bytes)
d_type      dc.b  $00               ;Device Type
d_stype     dc.b  $00               ;Device Subtype
d_version   dc.w  $00               ;Version Word

;******************************************************
```

SCANNING FOR 'STILL LIFE' FORMS, CAPTAIN

Now that we've found the scanner, we're ready to plant our thoughts in it. We do this by sending a few commands to the scanner, telling it what type of image we expect and what the scanner should do with the image before transferring it to us.

WE ARE ONE—OUR THOUGHTS ARE YOUR THOUGHTS

First, we send the scanner the halftone filter we want to use; then we set our scan window.

Halftone filter. Since we're going to do a halftone scan in our example, we issue a call to set the halftone filter. Note that we don't need to set this halftone filter if we choose to use one of the default filters or if we are going to scan in Line Art mode.
A halftone filter is nothing more than a defined threshold for each pixel of a 4 by 4 block. As the image under the mask changes intensity, the filter causes more or fewer of the dots to be black; the rest of the dots are white. The 4 by 4 block then becomes darker or lighter depending on the number of dots that are set to white within it, simulating gray tones even though our graphic mode knows only black and white.

Setting the halftone filter is easy; picking the filter pattern that best suits your needs is harder. Use one of the built-in patterns unless you have a better one. We use a simple Bayer type filter for this example. See Figure 1 and Code Sample 3.

0 $08	**8** $88	**2** $28	**10** $A8
12 $C8	**4** $48	**14** $E8	**6** $68
3 $38	**11** $B8	**1** $18	**9** $98
15 $F8	**7** $78	**13** $D8	**5** $58

Figure 1
Simple Bayer Pattern

```
;*******************************************************
;
;     CODE SAMPLE 3
;
;     In this code segment, we issue an Apple Scanner SEND
;     command by using the Apple SCSI Card Generic SCSI
;     call ($2B). By so doing, we can send our halftone
;     filter to the scanner.
;
;*******************************************************
```

```
htone_filter

                                ;
                                ; Issue the call.
                                ;
            lda   scan_dnum
            sta   dev_num3

            jsr   call_card
            dc.b  $04           ;Control Call Command
Number
            dc.w  cmd_list3
            rts

;*****************************************************

cmd_list3   dc.b  $03           ;PCount = 3
dev_num3    dc.b  $04           ;Device number
            dc.w  filter_data   ;Pointer to data
            dc.b  $2B           ;Control Code

;*****************************************************

filter_data                     ;Our Data
            dc.w  24            ;Total Length of Parms
            dc.l  send_fltr     ;CDB Pointer (Long)
            dc.l  DCData3       ;DCMove Ptr (Long)
            dc.l  $00000000     ;Rqst Sense Ptr (Long)
            dc.b  $00           ;Reserved
            dc.b  $00           ;SCSI Status
            dc.b  $00           ;Command Count
            dc.l  $00000011     ;Trans Count (Long)
            dc.b  $00           ;DMA Mode
            dc.l  $00000000     ;Reserved (Long)

;*****************************************************
```

```
send_fltr   dc.b  $2A              ;Scanner SEND Command
            dc.b  $00              ;Reserved
            dc.b  $02              ;Transfer Type
            dc.b  $00              ;Reserved
            dc.b  $00              ;Reserved
            dc.b  $02              ;Transfer ID Byte
            dc.b  $00              ;Reserved
            dc.b  $00              ;Transfer Length (High)
            dc.b  $11              ;Transfer Length (Low)
            dc.b  $00              ;Reserved

;******************************************************

DCData3     dc.l  send_data        ;Scanner SEND Data Ptr
            dc.l  $00000011        ;Transfer Count
            dc.l  $00000000        ;Offset
            dc.l  $00000000        ;Reserved

            dc.l  $00000000        ;DCStop
            dc.l  $00000000        ;Reserved
            dc.l  $00000000        ;Reserved
            dc.l  $00000000        ;Reserved

;******************************************************

send_data   dc.b  $44              ;4 X 4 Matrix Size
            dc.b  $08              ;Pel 0
            dc.b  $88              ;Pel 1
            dc.b  $28              ;Pel 2
            dc.b  $A8              ;Pel 3
            dc.b  $C8              ;Pel 4
            dc.b  $48              ;Pel 5
            dc.b  $E8              ;Pel 6
            dc.b  $68              ;Pel 7
            dc.b  $38              ;Pel 8
            dc.b  $B8              ;Pel 9
            dc.b  $18              ;Pel 10
            dc.b  $98              ;Pel 11
            dc.b  $F8              ;Pel 12
            dc.b  $78              ;Pel 13
            dc.b  $D8              ;Pel 14
            dc.b  $58              ;Pel 15

;******************************************************
```

Our scan window. Now that the scanner knows what halftone filter to use, we need to describe the scan window through which we'll view the document. Because we're using one of the Apple IIe graphics modes, our window will be fairly small. At 75 dpi in HiRes mode, or 150 dpi in Double HiRes mode, our window is about 3.75 inches across.

For the vertical screen, we have 192 pixels. At 75 dpi, our window is about 2.5 inches tall.

By using 75 dpi for HiRes and 150 dpi for Double HiRes, we can maintain a good aspect ratio. This allows us to display an image with minimum distortion.

In our example we use Double HiRes, so we first set the resolution for the X axis to 150 dpi and for the Y axis to 75 dpi. Then, we set our scan window's upper-left corner to absolute zero. See Code Sample 4.

```
;*******************************************************
;
;     CODE SAMPLE 4
;
;     In this code segment, we issue an Apple Scanner
;     DEFINE WINDOW PARAMETERS command by using the Apple
;     SCSI Card Generic SCSI call ($2B). This command
;     defines the area of the scanner glass we want to scan.
;
;*******************************************************

def_window
                              ;
                              ; Issue the call.
                              ;
            lda   scan_dnum
            sta   dev_num4

            jsr   call_card
            dc.b  $04         ;Control Call Command Number
            dc.w  cmd_list4
            rts

;*******************************************************
```

You should let users adjust the settings for Brightness, Threshold, and Contrast so they can customize the scan to the type of image being scanned (black and white or color; printed page or photo). If you let users choose Line Art or Grayscale, they can also optimize the scan for text or for an image. •

```
cmd_list4     dc.b  $03               ;PCount = 3
dev_num4      dc.b  $00               ;Device number
        dc.w  def_wndo          ;Pointer to data
        dc.b  $2B               ;Control Code

;*****************************************************

def_wndo                              ;Our Data
        dc.w  24                ;Total Length of Parms
        dc.l  def_wnd_cmd       ;CDB Pointer (Long)
        dc.l  DCData4           ;DCMove Ptr (Long)
        dc.l  $00000000         ;Rqst Sense Ptr (Long)
        dc.b  $00               ;Reserved
        dc.b  $00               ;SCSI Status
        dc.b  $00               ;Command Count
        dc.l  8+40              ;Trans Count (Long)
        dc.b  $00               ;DMA Mode
        dc.l  $00000000         ;Reserved (Long)

;*****************************************************

def_wnd_cmd   dc.b  $24               ;Scanner Define
Window
                                ;Parameters Command
        dc.b  $00               ;Reserved
        dc.b  $00               ;Reserved
        dc.b  $00               ;Reserved
        dc.b  $00               ;Reserved
        dc.b  $00               ;Reserved
        dc.b  $00               ;Transfer Length (High)
        dc.b  $00               ;Transfer Length (Mid)
        dc.b  8+40              ;Transfer Length (Low)
        dc.b  $80               ;Apple Bit

;*****************************************************

DCData4 dc.l  wndo_data         ;Scan Window Data Ptr
        dc.l  8+40              ;Transfer Count
        dc.l  $00000000         ;Offset
        dc.l  $00000000         ;Reserved

        dc.l  $00000000         ;DCStop
        dc.l  $00000000         ;Reserved
        dc.l  $00000000         ;Reserved
        dc.l  $00000000         ;Reserved
```

```
;*****************************************************
;     NOTE: Remember that all values longer than 1 byte
;     are in reverse order from native 65xxx code.
;*****************************************************

wndo_data   dcb.b 6,$00             ;Reserved
            dc.b  $00               ;Transfer Length (High)
            dc.b  40                ;Transfer Length (Low)

            dc.b  $01               ;Window Identifier
            dc.b  $00               ;Reserved

            dc.b  $00               ;X Resolution (High)
            dc.b  150               ;X Resolution (Low)

            dc.b  $00               ;Y Resolution (High)
            dc.b  75                ;Y Resolution (Low)
                                    ;
                                    ; We will use the corner as
                                    ; our upper-left position.
                                    ; This is at coordinate 0,0.
                                    ;
            dc.b  $00               ;Upper Left X (High)
            dc.b  $00               ;Upper Left X (Mid High)
            dc.b  $00               ;Upper Left X (Mid Low)
            dc.b  $00               ;Upper Left X (Low)

            dc.b  $00               ;Upper Left Y (High)
            dc.b  $00               ;Upper Left Y (Mid High)
            dc.b  $00               ;Upper Left Y (Mid Low)
            dc.b  $00               ;Upper Left Y (Low)
                                    ;
                                    ; Width is defined as the number
                                    ; of 1/1200-inch increments on
                                    ; the horizontal axis; must be on
                                    ; a byte boundary for both the
                                    ; start and end points. We will
                                    ; set for 4 inches and drop the
                                    ; extra.
                                    ;
            dc.b  $00               ;Width (High)
            dc.b  $00               ;Width (Mid High)
            dc.b  4*1200/256        ;Width (Mid Low)
            dc.b  4*1200            ;Width (Low)
                                    ;
                                    ; Length is defined as the number.
```

```
                                ; of 1/1200-inch increments on
the
                                ; vertical axis. We want ≈ 2-1/2
                                ; inches (or 3072 increments).
                                ;
        dc.b    $00             ;Length (High)
        dc.b    $00             ;Length (Mid High)
        dc.b    3072/256        ;Length 2.56*1200 (Mid Low)
        dc.b    3072            ;Length 2.56*1200 (Low)

        dc.b    $80             ;Median Brightness
        dc.b    $80             ;Median Threshold
        dc.b    $80             ;Median Contrast
        dc.b    $01             ;Image Composition (Halftone)
        dc.b    $01             ;Bits per Pixel
        dc.b    $00             ;Halftone Mask Always $00 (High)
        dc.b    $02             ;Downloaded Mask Pattern (Low)

        dc.b    $03             ;Padding Type
        dcb.b 2,$00             ;Reserved
        dc.b    $00             ;Compression Type (None)
        dcb.b 7,$00             ;Scanner Ref. is wrong
                                ; should be 7,
                                ; not 5.

;******************************************************
```

ENGAGE SCANNER

After telling the scanner how to scan, we need to tell it to start scanning. See Code Sample 5.

```
;******************************************************
;
; CODE SAMPLE 5
;
; This code segment issues an Apple Scanner SCAN
; command by using the Apple SCSI Card Generic SCSI
; call ($2B). This starts the actual scanning.
;
;******************************************************
```

```
start_scan
                              ;
                              ; Issue the call.
                              ;
            lda   scan_dnum
            sta   dev_num5

            jsr   call_card
            dc.b  $04         ;Control Call Command Number
            dc.w  cmd_list5
            rts

;******************************************************

cmd_list5   dc.b  $03         ;PCount = 3
dev_num5    dc.b  $00         ;Device number
            dc.w  scan_cmd    ;Pointer to data
            dc.b  $2B         ;Control Code

;******************************************************

scan_cm                       ;Our Data
            dc.w  24          ;Total Length of Parms
            dc.l  do_scan     ;CDB Pointer (Long)
            dc.l  DCData      ;DCMove Ptr (Long)
            dc.l  $00000000   ;Rqst Sense Ptr (Long)
            dc.b  $00         ;Reserved
            dc.b  $00         ;SCSI Status
            dc.b  $00         ;Command Count
            dc.l  $00000001   ;Trans Count (Long)
            dc.b  $00         ;DMA Mode
            dc.l  $00000000   ;Reserved (Long)

;******************************************************

do_scan     dc.b  $1B         ;SCAN
                              ;Parameters Command
            dcb.b 3,$00       ;Reserved
            dc.b  1           ;Transfer Length (Low)
            dc.b  $00         ;Wait and Home Bits = 0

;******************************************************
```

```
DCData5 dc.l  window_ID         ;Scan Window ID Ptr
        dc.l  1                 ;Transfer Count
        dc.l  $00000000         ;Offset
        dc.l  $00000000         ;Reserved

        dc.l  $00000000         ;DCStop
        dc.l  $00000000         ;Reserved
        dc.l  $00000000         ;Reserved
        dc.l  $00000000         ;Reserved

;*****************************************************

window_ID     dc.b  $01               ;Window Identifier

;*****************************************************
```

ENERGIZING!

We can get data from the scanner in two ways. We could get it all at once and then manipulate it to go on the screen. In our example, we would need a buffer with 115,200 pixels or 14,400 bytes for the data: (4.0 inches * 150 dpi horizontally) * (2.56 inches * 75 dpi vertically).

To save the amount of RAM our program uses, however, we set up a buffer large enough for only one line; then we read each line from the scanner and display it until the entire image is on the screen. See Code Sample 6.

The data returned by the scanner is 8 pixels per byte. Bit 7 is the left-most pixel and bit 0 is the right-most pixel; a value of 1 means a black dot in the image. In the Apple II HiRes mode, we have 7 pixels per byte. Bit 0 is the left-most pixel and bit 6 is the right-most pixel; a value of 1 means a white dot. Because the formats are different, the program must convert the returned data, which it does as it goes, using code shown in Code Sample 6.

```
;*****************************************************
;
; CODE SAMPLE 6
;
; In this code segment, we issue a series of calls to
; the Apple Scanner by using the Apple SCSI Card Generic
; SCSI call ($2B). We first issue a GET DATA STATUS
; call to see if there is enough data. Then we read
; in a single scan line with a READ call. The data is
; then converted and placed in a video buffer.
;
;*****************************************************
```

```
get_data
            stz   scan_line   ;Init the scan line to 0.
                              ;
                              ; Issue the call.
                              ;
            lda   scan_dnum
            sta   dev_num6
            sta   dev_num65

@get_data2  jsr   call_card
            dc.b  $04         ;Control Call Command Number
            dc.w  cmd_list6
            bcs   @out
                              ;
                              ; Is there enough data?
                              ; Enough data = 1 scan
                              ; line of 4 inches at 150
                              ; dpi (or 600 pixels). At
                              ; 8 pixels per byte, the
                              ; data will be padded to
                              ; 75 bytes.
                              ;
            lda   scan_data
            bne   @have_line
            lda   scan_data+1
            bne   @have_line
            lda   scan_data+2
            cmp   #rqst_cnt   ;Decimal 75
            blt   get_data
                              ;
                              ; We have the data. Read
                              ; it.
                              ;
@have_line  jsr   call_card
            dc.b  $04         ;Control Call Command Number
            dc.w  cmd_list65
            bcs   @out
                              ;
                              ; Now we need to invert
                              ; the data.
                              ;
```

```
            lda   #80          ;80 bytes/line for Double HiRes

            sta   byte_count
            stz   byte_index
@loop_1     lda   #$07
            sta   seven        ;Pixels/byte
@loop_2     ldx   #rqst_cnt-2
            asl   raw_image+\
            rqst_cnt-1         ;Shift bits out the top to
@loop_3     rol   raw_image,x :the next byte 1 at a time
            dex
            bpl   @loop_3
            ldx   byte_index   ;Shift the last bit into
            ror   screen,x     ;this byte. This reverses the
            dec   seven        ;bit ordering and takes 8 bits
            bne   @loop_2      ;per byte down to 7.
            lsr   screen,x
            inc   byte_index
            dec   byte_count
            bne   @loop_1
                              ;
                              ; Move data to scan line.
                              ;
            ldx   scan_line
            jsr   on_screen
            inc   scan_line
            bra   @get_data2

@out        lda   #$00
            clc
            rts

;******************************************************

scan_line   dc.b  $00          ;Scan Line Index
byte_count  dc.b  $00          ;Number of bytes left
byte_index  dc.b  $00          ;Current Byte in use
seven       dc.b  $00          ;Count off 7 pixels
screen      dcb.b 80,0         ;Place to do the screen

;******************************************************

cmd_list6   dc.b  $03          ;PCount = 3
dev_num6    dc.b  $00          ;Device number
            dc.w  gd_status    ;Pointer to data
            dc.b  $2B          ;Control Code
```

```
cmd_list65  dc.b  $03          ;PCount = 3

dev_num65   dc.b  $00          ;Device number
            dc.w  read         ;Pointer to data
            dc.b  $2B          ;Control Code

;******************************************************

gd_status                      ;Our Data
            dc.w  24           ;Total Length of Parms
            dc.l  get_stat     ;CDB Pointer (Long)
            dc.l  DCData6      ;DCMove Ptr (Long)
            dc.l  $00000000    ;Rqst Sense Ptr (Long)
            dc.b  $00          ;Reserved
            dc.b  $00          ;SCSI Status
            dc.b  $00          ;Command Count
            dc.l  $0000000C    ;Trans Count (Long)
            dc.b  $00          ;DMA Mode
            dc.l  $00000000    ;Reserved (Long)

read                           ;Our Data
            dc.w  24           ;Total Length of Parms
            dc.l  get_data2    ;CDB Pointer (Long)
            dc.l  DCData65     ;DCMove Ptr (Long)
            dc.l  $00000000    ;Rqst Sense Ptr (Long)
            dc.b  $00          ;Reserved
            dc.b  $00          ;SCSI Status
            dc.b  $00          ;Command Count
            dc.l  rqst_cnt     ;Trans Count (Long)
            dc.b  $00          ;DMA Mode
            dc.l  $00000000    ;Reserved (Long)

;******************************************************

get_stat    dc.b  $34          ;GET DATA STATUS
                               ;Parameters Command
            dcb.b 7,$00        ;Reserved
            dc.b  12           ;Transfer Length (Low)
            dc.b  $00          ;Wait and Home Bits = 0
```

```
get_data2   dc.b  $28          ;READ

                              ;Parameters Command
            dcb.b 4,$00       ;Reserved
            dc.b  $01         ;Window ID
            dc.b  $00         ;Transfer Length (High)
            dc.b  $00         ;Transfer Length (Mid)
            dc.b  rqst_cnt    ;Transfer Length (Low)
            dc.b  $00         ;Wait and Home Bits = 0

;*****************************************************

DCData6     dc.l  data_cnt    ;Data Pointer
            dc.l  12          ;Transfer Count
            dc.l  $00000000   ;Offset
            dc.l  $00000000   ;Reserved

            dc.l  $00000000   ;DCStop
            dc.l  $00000000   ;Reserved
            dc.l  $00000000   ;Reserved
            dc.l  $00000000   ;Reserved

DCData65    dc.l  raw_image   ;Data Pointer
            dc.l  rqst_cnt    ;Transfer Count
            dc.l  $00000000   ;Offset
            dc.l  $00000000   ;Reserved

            dc.l  $00000000   ;DCStop
            dc.l  $00000000   ;Reserved
            dc.l  $00000000   ;Reserved
            dc.l  $00000000   ;Reserved

;*****************************************************

data_cnt                      ;Data Space
            dcb.b 2,$00       ;Reserved
            dc.b  $00         ;Data Length
            dc.b  $00         ;Block
            dc.b  $00         ;Window Identifier
            dcb.b 4,$00       ;Reserved
scan_data   dc.b  $00         ;Scan Data (High)
            dc.b  $00         ;Scan Data (Mid)
            dc.b  $00         ;Scan Data (Low)

raw_image   dcb.b 100,$00     ;Scanned Data Image

;*****************************************************
```

PUT IT ON THE SCREEN, ENSIGN

Because we display the image in black and white, we need to set up the graphic soft switches accordingly. In our example, we display our image in HiRes Page 1, and we assume black and white display. On a color video monitor, the image would appear in black and white. See Code Sample 7.

```
;*******************************************************
;
;     CODE SAMPLE 7
;
;     In this code segment, we toggle the HiRes soft
;     switches so that we can see what was just scanned.
;
;*******************************************************

display
                              ;
                              ; Save the current state.
                              ;
            lda   RDTEXT
            sta   @text       ;Text/Graphics
            lda   RDMIX
            sta   @mixed      ;Mixed?

            lda   RDPAGE2
            sta   @page       ;Page 1 or 2

            lda   RDHIRES
            sta   @hires      ;HiRes Mode?

            lda   RD80VID
            sta   @80col      ;80-Column Mode?

            sta   SET80VID    ;Set 80-Column Mode
            sta   TXTCLR      ;Standard Apple II Graphics
            sta   MIXCLR      ;Clear Mixed Mode
            sta   TXTPAGE1    ;Page 1
            sta   HIRES       ;HiRes Mode
            sta   CLRAN3      ;Clear annunciator 3
```

```
            sta   KBD_STRB    ;Clear Key Strobe

@key_loop   lda   KBD         ;Get key
            bpl   @key_loop   ;Wait for Key Press
            sta   KBD_STRB    ;Clear Key Strobe
            cmp   #ESC        ;ESC Key
            clc
            bne   @chk_txt
            sec               ;Exit on ESC

            lda   SETAN3      ;Set annunciator 3.

@chk_txt    lda   @text
            bpl   @chk_mix
            sta   TXTSET      ;Text on

@chk_mix    lda   @mixed
            bpl   @chk_page
            sta   MIXSET      ;Mixed on

@chk_page   lda   @page
            bpl   @chk_hires
            sta   TXTPAGE2    ;Page 2

@chk_hires  lda   @hires
            bmi   @chk_40col
            sta   LORES       ;HiRes Off

@chk_40col  lda   @80col
            bmi   @rts
            sta   CLR80VID    ;80-Column on

@rts        rts

@text       dc.b  $00
@mixed      dc.b  $00
@page       dc.b  $00
@hires      dc.b  $00
@80col      dc.b  $00

;*******************************************************
```

FILE THE REPORT AND HEAD FOR HOME

Now, save the image in its displayable format. Save it as you would any file, using standard ProDOS MLI calls.

FINAL LOG ENTRY

The ability to bring printed images into the computer opens up many possibilities for you and for your customers. Programs that use graphics can import and add color to printed images. For example, users can put together files that include family photos. These files can then be transmitted electronically to others for viewing.

You can also give users control over a number of scan parameters. For example, you could allow them to position the scan window on a graphic representation of the scanner glass; users could then position the scan without adjusting the printed page on the scanner glass. Or you could allow users to specify the resolution of the scan, showing them how the scan window size changes.

Although not demonstrated here, Line Art mode provides very clean images of scanned text. If you use Line Art mode to support optical character recognition (OCR), users can import text and avoid retyping entire manuscripts.

The possibilities are endless. Have fun exploring them. That is, after all, what it is all about—doing more with your Apple II and having fun doing it.

Thanks to Our Technical Reviewers
Greg Banks, Charlie Eckhaus, Dave Lyons, Llew Roberts, Mike Seilnecht •

Zz speaks

PRINT HINTS WITH LUKE & Zz

Once upon a time, in an engineering department far, far away, the great implementors (GIs) ran across a problem with their Frankensteinian monster. They had designed and built a powerful beast. A beast that had traveled great distances into unknown territories, and that had cut many new paths in the hot lead jungle. Many spoke of the creation, and it had become well known and respected in many lands and platforms. But the complexity that only the great implementors had dared to conceive was being challenged by the great hackers of the land. Many of them had conquered the secrets of the beast, and shared the knowledge of the great implementors. Soon, even mortal hackers would be able to control the beast. This would not do. . .

The GIs traveled far to find the infamous "field workers." They had already given the GIs the gift of dynamic, incomprehensible data structures, and the GIs were hopeful that they could provide something more. They could not. . .

At last, on their return from the field, they ran across a court jester. Like all good jesters, he was trying hard to find some fun. He tried to get the GIs to play a game with him. He tried Monopoly, Pictionary, even checkers, but the GIs were not interested (he did notice a little glimmer when he mentioned thermonuclear war, but not enough to be important). As a last resort, he finally said, "Pick a number." Suddenly, the great gods of arbitration came down on the GIs like a 5-ton weight. "Pick a number!" one of them shouted. "It's beautiful!" laughed another one. And soon they were off to the lab to implement their new discovery.

This folks, is the not-based-on-a-true-story story of the constant named `iPfMaxPgs`. This constant is part of the Printing Manager, and limits the number of pages that can be spooled (that is, written to disk) in one print job. In the old days, when even engineers at Apple were using floppy drives, the Print Shop came to the conclusion that there should be a limit on the number of pages you could print in a single job. If not, the user could easily use up all the disk space on the boot disk, which at the time, led to other, even more interesting problems. The decision to limit the number of pages was quickly followed by a decision to use the court jester's advice and "pick a number." The magic number is 128, and is referenced by the constant `iPfMaxPgs`.

So now, you are printing your favorite report, all 130 pages of it, and just to annoy your neighbors, you're printing to your ImageWriter II in Best mode. You've placed the printer next to the window in the bathroom where the acoustics are especially outgoing, and the window is open for the extra cooling effect (at least, that's what you told the neighbors). As the head rips across the bottom of page 128 like nails across the chalkboard, you are suddenly greeted with a Printing Manager error code. Sure, your neighbors are happy, but you still have two pages to go. You frantically dig for that suicide prevention number (that your neighbors have obviously borrowed), but decide instead that printing more than 128 pages in one job would be a great way to get new neighbors.

To print jobs longer than 128 pages, you simply treat them like multiple jobs, printing each set of 128 pages just like the first set. To make this even simpler, it would be nice if there was one line of code that told you when to close the current document and open the new one. Welcome to the MOD squad (or the % Club for you C dudes).

The MOD instruction is very useful for tracking sets of things, like the number of pages in a print job. Most people that have implemented a Macintosh printing loop (you can spot them by the gray hair and growing forehead) use some kind of page loop.

The following code fragments will show the flow of control using the MOD operator. The parameters of the Toolbox routines have been omitted for simplicity.

Before you knew enough words to type 128-page documents, you used to have a print loop that figured out how many pages you had to print and then printed them one at a time.

This works well for bothering neighbors, but you'll never get any For Sale signs with this technique. To print jobs larger than 128 pages, you need to call **`PrOpenDoc/PrCloseDoc`** within the FOR loop. Whenever you reach 128 pages, you need to call **`PrCloseDoc`** to close the current document, and then **`PrPicFile`** to despool the (now full) spool file. Once this is done, a new document is opened, and everything continues as before.

```
FOR pageIndex := firstPage
      TO kNumberOfPages - 1 DO
BEGIN
      (* If we are on page number
       128, we need to close and
       reopen the document *)
      IF(pageIndex - firstPage)
      MOD iPFMaxPgs = 0 THEN
      BEGIN
            (* Make sure there is a
                  document open before
                  calling PrCloseDoc! i.e.
                  if this is the first
                  page, don't call
                  PrCloseDoc... *)
            IF pageIndex <>
                  firstPage THEN
            BEGIN
                  PrCloseDoc(...);
                  PrPicFile(...);
            END;
            (* Now open a fresh, 128
                  page spool file. *)
            PrOpenDoc(...);
      END;
      (* Call PrOpenPage/
       PrClosePage for each
       page of the document. *)
      PrOpenPage(...);
      PrintAPage(...);
      PrClosePage(...);
END;
(* Finally, close the document
      and despool the spool file.*)
PrCloseDoc(...);
PrPicFile(...);
.
.
.
```

Pretty straightforward, but you'd be surprised how many developers the great implementors have caught with this one. Using the above method, you can safely print large documents on any device, without having to worry about overflowing the spool file. As for your neighbors, you could always put a Pause/Continue button in your Printing Status dialog, but then, where's the fun in that?

PALETTE MANAGER ANIMATION

In the last several weeks many of you have asked "How do I animate colors with the Color Manager?" I usually answer, "The Color Manager is not a good way to animate colors. Try the Palette Manager instead." I figure that for every person who asks this question there are a hundred others out there trying to figure out the answer by themselves. This article is for all of you independent types who never ask questions but could use the answer just the same.

RICH COLLYER

This article comes with a sample: GiMeDaPalette. The article shows how to do color table animation by using the Palette Manager, and by the end of the article I think you'll be convinced that using the Palette Manager is the only way to fly.

To see the animation effect, you will need to run the sample on a monitor (device) that uses a color look-up table, often called a clutType device; you need a color look-up table to do color table animation.

The sample is designed to run only under System 6.0.5 or greater, or when 32-Bit QuickDraw is installed; the Palette Manager shows its abilities best in these environments. This limitation may discourage those who want their applications to work on all systems and hardware configurations, but remember that if you stick to 32-Bit QuickDraw or 6.0.5, you'll be able to take advantage of the upgrades and improvements that Apple provides. If you don't, your application is likely to stagnate. One possible way to work around this dilemma is to separate your code into a pre-32-Bit QuickDraw version and a 32-Bit QuickDraw version. This will make your application more complex, but it will give you the flexibility to work under most, if not all, color systems.

When you run the sample application, you will find that there is a File menu, which just allows you to quit, and a Palette menu that allows you to pick the color usage of the palette. The sample initializes itself to use a palette with the color usage Courteous.

RICH COLLYER Eagle Scout/Fractal Hacker, claims there is nothing unusual about himself. He's done the routine, everyday stuff we all do, such as attending a sacrifice at a temple in the Himalayas, climbing a 17,500-foot peak (climbing the 20,000-foot one would have been "unreasonable"), and strolling the byways of Katmandu and Bhutan. His adventurous tendencies led him to pursue a physics degree from Cal Poly with a specialty in computational fluid dynamics, and routinely compels him to climb rocks. He's survived at least three 20-foot falls; outwardly he seems unscathed, but we have to wonder. He has a distinct penchant toward chaos, named his dog Precious of Bonshaw, and fosters a burning desire to be a DTS engineer. Nothing unusual. •

When you want to animate the palette, you will want to select one of the menus that contains the word animated. (Animated, Tolerant + Animated, Explicit + Animated, Tolerant + Explicit + Animated).

A LOOK AT THE SAMPLE

The three major parts of the sample:

- Setting up the color environment
- Picking a color for drawing
- Animating the colors will be needed in most applications that require color table animation.

SETTING UP THE COLOR ENVIRONMENT

The four main lines of code that I used to set up the environment in GiMeDaPalette are

```
mycolors = GetCTable(clutID);

(*mycolors)->ctFlags |= 0x4000;

srcPalette = NewPalette(numcolor, mycolors, pmCourteous, 0);

SetInhibited(pmCourteous);
```

Since this sample is a more general use of the Palette Manager, I started by building a palette that is Courteous. This means that you get what colors the system can give you with the best matches it can make, but the Palette Manager is not going to

PALETTE MANAGER HISTORY

1987 – The Palette Manager arrived on the scene late in the development of the original Macintosh II. The engineer who was responsible for the Palette Manager was given only a few weeks to produce it. Under the circumstances, it's amazing that the Palette Manager worked at all.

1988 – System 6.0.2 shipped with a new version of the Palette Manager that was much closer to the way it was supposed to work, but it still didn't do all that people wanted it to.

1989 – The first version of 32-Bit QuickDraw included a lot of big changes and improvements to the Palette Manager. There were still a few problems, but the Palette Manager was finally able to do what people really wanted it to.

1990 – The 32-Bit QuickDraw version of the Palette Manager was made part of the system software in 6.0.5. The last of the major problems were ironed out, and the new Palette Manager was available to everyone who ran system 6.0.5 or greater, with or without 32-Bit QuickDraw.

change any of the colors in the environment to give you what you want. The palette is built with a color table that is stored as a resource and retrieved with the trap call to **GetCTable**. The manipulation to the **ctFlags** is described in a sidebar, "Other Features of the Palette Manager," later in this article; basically the manipulation makes it possible to use the colors in an offscreen world. The palette has 256 colors (**numcolor** = 256) and the tolerance is set to zero. The tolerance value tells the Palette Manager how close the color match needs to be, but unless you have the usage set to **pmTolerant**, it is ignored. For more information on these calls, see the Palette Manager chapters in *Inside Macintosh* (volume V, chapter 7, and volume VI, chapter 20). My function **SetInhibited**, is described in the sidebar, "Other Features of the Palette Manager." I used it to set the palette usage so that the palette will be good for any pixel depth that the window may end up on.

After setting up the palette, you attach it to the window:

```
SetPalette ((WindowPtr) myWindow, srcPalette, TRUE);
```

If you want more control over when the window gets its updates, you would want to use **NSetPalette** instead of **SetPalette**. **NSetPalette** allows you to specify whether you want the updates to happen only when the window is in the background, only when it is in the foreground, always, or never.

The simplest way of using palettes is to store them as resources of type **'pltt'**. When **GetNewCWindow** is called the system looks for a **'pltt'** resource with the same ID as the window being opened. If it is found, the palette is loaded and attached to the window. It is also possible to have a palette that is used for all the windows an application may open; in this case when **NewCWindow** or **GetNewCWindow** is called (and no **'pltt'** with the same ID is found) the **'pltt'** with ID = 0 is used.

When you select one of the Animated menus, the code calls **SetInhibited** and passes the function a usage of **pmAnimated**. The function then sets up the palette to have the new usage and makes the palette available for animation.

PICKING A COLOR FOR DRAWING

In GiMeDaPalette I don't have to pick a color to draw with. If you want to do any drawing in your application, other than calling **CopyBits**, you call the trap **PmForeColor** to select a color to draw with and then just draw.

ANIMATING COLORS

Once you select one of the Animated menus, GiMeDaPalette requires only four lines of code to animate colors. First you save the first color in the palette:

```
GetEntryColor(srcPalette, 1, &changecolor);
```

Then you cycle the colors 2 to 254:

```
AnimatePalette(myWindow, StoreCTab, 2, 1, numcolor - 2);
```

Next you move the saved color to the last entry :

```
AnimateEntry(myWindow, numcolor - 1, &changecolor);
```

Finally you save the new version of the palette into the color table for use during the next animation:

```
Palette2CTab(srcPalette, StoreCTab);
```

The sample does not animate the entire palette, because the Palette Manager will not let you animate white (entry 0) and black (entry 255).

USING THE COLOR MANAGER

All of this can be done with the Color Manager, but to do it and get the same functionality, you will need to generate considerably more code. The main problem with the Color Manager is that it does not provide automatic support for multiple monitors, color arbitration, compatibility with other applications, and several other features that are basic to any real color application.

OTHER AUTOMATIC FEATURES OF THE PALETTE MANAGER

KEEP IT SIMPLE

The Palette Manager takes care of several details for you, and the result is a simpler, more elegant application. It also provides more compatibility; that is, it helps ensure that the application will work under GC QuickDraw and will be friendly under MultiFinder.

TAKE CARE OF THE ENVIRONMENT

Like a good citizen in a community, an application must take care of its environment. If applications have no concern for their environments, users will constantly need to reboot their machines to get back to stable ground. The Palette Manager provides this support without extra code.

Background. One of the differences between the Color Manager and the Palette Manager is that the Color Manager usually animates the background and the Palette Manager tries not to. The Palette Manager reserves the colors so that no other applications can use them. This means that when the colors are animated, only the foreground animates, if possible. In general the Palette Manager tries to make sure that the colors your application is using will not make a mess of other applications that are running at the same time.

Reserve entries. You can reserve the colors with the Color Manager by using **ReserveEntry**, but if you do, you can no longer use **Index2Color** to get the colors you need. But even if you get around that problem, you still cannot call **RGBForeColor** to set the color; you need to change the graphics port directly, and we all know what a *bad* idea this is. Once again the Palette Manager is cleaner, because it does not force your application to manipulate the graphics port.

MultiFinder switch. The Palette Manager cleans up when you are switched out in MultiFinder. If another application that uses a different color environment is running at the same time as yours, it gets the colors it wants when it is the frontmost application, and you get the colors you want when you are the frontmost application. The Color Manager does not do this for you; you must do the work yourself.

Multiple monitors. The Palette Manager will work over multiple monitors, while the Color Manager does not. If you wish to run your application on multiple monitors, and you are using the Color Manager, you will have to worry about what monitors your windows span and make sure that **SetEntries** is called for each monitor. This can get really cumbersome.

GIVE ME SPEED

Generally you would expect the Color Manager to run faster, since the **AnimatePalette** trap ultimately calls **SetEntries**. However, the Palette Manager is just as fast as the Color Manager. And don't forget that while the Palette Manager is just as fast as the Color Manager, it is also providing the support for all of its cool features; the Color Manager provides none of this support.

GetEntries: ONE OF THE GOTCHAS IN THE COLOR MANAGER

Some people think they can call **GetEntries** to save the current color table and later rebuild the color table with **SetEntries**. Unfortunately, **GetEntries** and **SetEntries** do not work well together. Each time you call **GetEntries** and follow it with **SetEntries**, the color table will get a little lighter. What happens is that when you call **SetEntries**, the colors are gamma corrected and **GetEntries** does not reverse this effect. So each gamma correction causes the color table to get lighter. For more information about gamma correction take a look at "Designing Cards and Drivers for the Macintosh Family."

USE THE PALETTE MANAGER

I hope that I have convinced you that the Palette Manager is considerably better at color table animation than the Color Manager. Let's go over the highlights:

- The Palette Manager makes your application more compatible with GC QuickDraw and MultiFinder. The Color Manager does not do this for you.
- The Palette Manager protects your application from making a mess of the color environment by not letting it animate black and white. With the Color Manager you have to protect yourself.

Table 1
Palette Manager versus Color Manager

	Palette Manager	Color Manager
Compatible with MultiFinder	☒	☐
Compatible with GC QuickDraw	☒	☐
Compatible with other color apps	☒	☐
Easy to use	☒	☐
Automatic color arbitration	☒	☐
Easy support of multiple monitors	☒	☐
Requires you to implement all features	☐	☒
Renders your app hardware dependent	☐	☒
Not worth the effort	☐	☒

- The Palette Manager arbitrates the colors in the color environment for you. This means that you don't have to worry about your window getting the colors it needs when it is the frontmost window. Color Manager does not do this for you.
- The Palette Manager takes care of multiple monitor support for you. Multiple monitor support comes free with the Palette Manager; there are no freebies in the Color Manager.
- The Palette Manager can cleanly reserve the entries that you wish to animate; there is no clean way to do this with the Color Manager.

INTERRUPTS You may have noticed that a call to **`AnimatePalette`** or **`SetEntries`** turns off interrupts. All Apple video cards, and most third-party video cards, turn off interrupts until the next VBL (vertical bLanking interrupt) when **`SetEntries`** is called. This generates a cleaner, smoother color animation, but it is a real headache for anyone who wants to animate colors and at the same time take in data on the serial lines or play a sound. Any kind of interrupt-driven process, which requires a lot of attention, is not going to work when **`SetEntries`** is being called repeatedly. •

OTHER FEATURES OF THE PALETTE MANAGER

`ctFlags` BIT 14

If you would like to use an offscreen world and to animate the images that you create, this is the bit for you. If you

- set bit 14 of the **`ctFlags`** field in your color table
- use this color table to pick your colors when drawing in the offscreen world
- set the usage of your palette to **`pmAnimated + pmExplicit`** and then call **`CopyBits or DrawPicture`**; the colors will map correctly and the image will animate. This feature allows an offscreen world full access to the color table via the index values. If you do not follow these three steps, you will get only a black-and-white image out of the **`DrawPicture`** or **`CopyBits`**. I have included this feature in the sample GiMeDaPalette. The code below is just clippings from the sample itself.

```
----------------
mycolors = GetCTable (clutID);
(*mycolors)->ctFlags |= 0x4000;
...
srcPalette = NewPalette (numcolor,
mycolors, pmAnimated + pmExplicit, 0);
...
err = NewGWorld (&offscreenGWorld, 8,
&WinMinusScroll, mycolors, nil, nil);
...
DrawPicture (ThePict, &WinMinusScroll);
```

`pmAnimated+pmExplicit`

This is one of the really cool features of the Palette Manager. Generally when you build a palette, the color matching will just put the colors where they fit best, but when you have a usage of explicit, the colors are placed in the exact index locations in which you want them. This is important because you can't set a palette to an offscreen world, and you generally want to be able to draw off screen first and then copy to the screen. This is a problem because you still need access to the palette colors. But you can achieve this access by building the palette in such a way that you know the colors are at the exact index values you have specified in your color table. This feature in conjunction with the **`ctFlags`** bit 14 feature will give you the access you need in your offsceen world. For more information about this feature take a look at the Palette Manager chapter in *Inside Macintosh*, volume VI (chapter 20).

`pmInhibit`

This usage option is yet another cool feature of the Palette Manager. It allows the application to specify which colors are to show up on which monitors. So if you have a palette of 256 colors, but the window is on a 4 bit/pixel screen, you'll get the 16 of the 256 colors that are labeled as **`pmInhibitC2`**. In my sample the function, **`SetInhibited`**, sets the first two colors to always show. The next 12 entries will be available only if the window is on a screen that is set to 4 or 8 bits/pixel. The rest of the colors will be available only if the window is on an 8 bit/pixel screen. You can also set the palette to consider whether the window is on a gray scale screen or not. For more information about this feature take a look at the Palette Manager chapter in *Inside Macintosh*, volume VI (chapter 20) and the sample code.

If this has convinced you that the Palette Manager is the way to go, I think you've taken a big step toward making the next great color application that Mac users around the world will love.

Thanks to Our Technical Reviewers
Guillermo Ortiz, Forrest Tanaka, David Van Brink, John Zap •

THE POWER OF MACINTOSH COMMON LISP

Macintosh Common Lisp (MCL) is a powerful implementation of the Lisp language as well as a dynamic development environment. This article describes major aspects of the MCL language and environment. It provides essential information for non-Lisp programmers who are unaware of the power of this language or of the MCL development environment, as well as for Lisp programmers who are unaware of MCL features or performance.

RUBEN KLEIMAN

As the price of memory plummets and powerful computers become as cheap as sand, developers are beginning to look afresh at the positive aspects of dynamic programming environments, like Lisp and Smalltalk, that once seemed too slow and memory-hungry. These environments offer the proven ability to generate and run large-scale applications, easily access the toolbox, and call MPW C, Pascal, or Assembler programs. With comprehensive and elegant class libraries for defining user interfaces, these environments promise to significantly improve programmer productivity over traditional languages.

Many Lisp environments are available for the Macintosh. We'll focus here on what one particular dynamic programming environment, Apple's own Macintosh Common Lisp (MCL), has to offer. We'll compare it to the programming environment provided by MPW in conjunction with MacApp, Apple's object-oriented application framework based on Pascal. We'll take a close look at its key advantages, and will illustrate them with fragments from a sample program. The entire sample program plus a step-by-step description of its development can be found on the *Developer Essentials* disc that accompanies this issue.

RUBEN KLEIMAN was hired as a Senior Research Scientist at Apple because of his proven ability to withstand large amounts of pain, function well after sleepless nights, and sincerely worry about things that don't need to be worried over. He studied physics and comparative literature during a college career that he likens to the Harrad Experiment; he was never totally convinced that his coed college in Florida (which had no nudity rules in the dorms or swimming pool areas) wasn't backed by the CIA. The experience drove him to Cambridge, Massachusetts, where he studied linguistics (Hittite, Sanskrit) at Harvard. He then began researching the universe (that is, the unified theory of quantum mechanics and relativity) while being remunerated by MCC, Computervision, and MITRE. Down to his last five dollars, he joined

WHAT IS MCL?

MCL is a powerful implementation of the Lisp language. (If you're new to Lisp, take a look at the sidebar "A Mini Lisp Tutorial" for a quick overview of how it differs from Pascal.) MCL provides full compatibility with the Common Lisp standard, an extensive object-oriented system, and a rapid prototyping development environment.

MCL 2.0 supports Common Lisp and the Common Lisp Object System (CLOS). This extension of the Common Lisp standard offers an object-oriented programming paradigm for Lisp, within which MCL implements a class library for developing user interfaces. The MCL environment includes a syntax-oriented text editor for Lisp; a direct way to navigate through sources; a tracer, stepper, and backtracer; and the ability to disassemble code just in case you want to shave off a microsecond.

The key advantages of MCL are its interactivity, the inherent power of symbolic processing in Lisp, the overall consistency of its object library, and its abstraction away from the Macintosh event-loop style of programming. We'll take a closer look at these advantages as we compare MCL with MacApp/MPW.

A COMPARISON OF MCL AND MACAPP/MPW

MCL and MacApp/MPW are both object-oriented programming environments available from Apple. We'll compare four different aspects of these environments:

- Their language bases
- Their class libraries and event systems
- Their strong points as development environments
- Their size and performance specifications

LANGUAGE BASES

MacApp is based on ObjectPascal, a set of object-oriented extensions to Pascal somewhat on a par with the C++ extensions to C. MCL is based on the Common Lisp standard (ANSI X3J13 Committee), which includes the Common Lisp Object System (CLOS), an object-oriented extension to Lisp. Table 1 gives an overview of what these languages offer.

The most striking differences in the languages are (1) their syntax (described in the sidebar "A Mini Lisp Tutorial"), (2) the ability of Common Lisp to deal with typeless variables, and (3) Common Lisp's automatic garbage collection. Let's turn our attention to the latter two differences.

Apple three years ago. In his spare time he plays together with his wife, sculpts, photographs, skateboards, is writing the Great Argentinian Novel (he's a native of Buenos Aires), and tries to cycle at least once a week. But he's a little worried that it might not make any difference until his next life. •

A beta release of Macintosh Common Lisp 2.0 should be available from APDA by the time you read this. See the inside back cover for information on how to contact APDA. •

A MINI LISP TUTORIAL

This tutorial gives a quick overview of Lisp, but it omits many things, like macros and other intimidating nested monsters. Excellent books on Lisp are available to suit most tastes: see the list of recommended reading at article's end.

Lisp belongs to the Functional Programming Language family. A key point is that every Lisp function or expression always returns a value—whether you want it to or not! It is thus difficult to talk about Lisp "programs" because there is no preferred "entry point" or "main."

Lisp functions differ from, say, Pascal functions in that in Lisp the function name is enclosed in parentheses along with its arguments:

Pascal	Lisp Equivalent
`SysBeep(120);`	`(SysBeep 120)`
`ResError;`	`(ResError)`

To assign a value to a global variable, you must declare the variable as global before using it and then use the assignment function **`setq:`**

Pascal	Lisp Equivalent
`VAR x: integer;`	`(defvar x nil)`
`...`	
`x := 2;`	`(setq x 2)`

Control statements like **`IF`** and **`CASE`** are available:

Pascal	Lisp Equivalent
`if x = 2`	`(if (= x 2)`
`then SysBeep(30)`	`(SysBeep 30)`
`else x := 0;`	`(setq x 0))`
`case x of`	`(case x`
`1: Sysbeep(30);`	`(1 (SysBeep 30))`
`2: x := 10;`	`(2 (setq x 10))`
`3: x := 20;`	`((3 4) (setq x 20)))`
`4: x := 20;`	

The simplest use of Common Lisp's powerful repeat control structure, called **`loop,`** is as follows:

Pascal	Lisp Equivalent
`x := 0;`	
`repeat`	`(loop for x from 1 to 5`
`x = x + 1;`	`do (SysBeep 30))`
`SysBeep(30);`	
`until x = 5`	

Function arguments are evaluated from left to right *before* they are actually passed to the function. The only exception to this is the function **`defmacro`** (and related ones) used to create macros.

The function **`quote`** helps you to pass a symbol or a list instead of passing whatever the symbol or list evaluates to. For example, if the symbol **`x`** is bound to 19 and **`foo`** is some function, then evaluating **`(foo x)`** will result in **`foo`** being passed the value 19, but **`(foo (quote x))`** will be passed the symbol **`x.`** A short form for **`quote`** is a single quotation mark: for example, **`(foo (quote x)) = (foo 'x)`**.

Table 1
Features of ObjectPascal Versus Common Lisp

Feature	ObjectPascal	Common Lisp
Instance variables	Yes	Yes
Class variables	No	Yes
Multiple inheritance	No	Yes
Inheritance types	One	One standard, user-redefinable
Method combination	Not applicable	Yes
Before/after methods	No	Yes
Methods on instances	No	Yes
Method discrimination	On single argument	On all arguments
Toolbox interface	Yes	Yes
Variable typing	Required	Optional
Garbage collection	Manual	Automatic
Foreign language interface	Yes (MPW object files)	Yes (MPW object files)
Error handling	Yes	Yes

In Lisp, you need not declare a variable's type. You can assign to a Lisp variable any type of object, or many types of objects at different times, within a lexical scope. The type information is associated with the data objects themselves rather than with the variables. However, declaration statements are available for optimal compilation. A common practice is not to type variables until the program is thoroughly debugged, and then to use typing only in the most crucial parts of the code. For better performance, you can require the run-time system to forego type checking.

Common Lisp does automatic garbage collection of inaccessible values (for example, objects, strings, arrays)—that is, values that are implicitly deallocated. A key advantage of this is simplification of your code. For example, the following statement allocates an instance of the class **`Window`** and binds it to the variable **`myWindow:`**

```
(setq myWindow (make-instance 'Window))
```

If thereafter you set **`myWindow`** to a different value, say,

```
(setq myWindow (make-instance 'Dialog))
```

Common Lisp will free up the space occupied by the **`Window`** instance (unless, of course, you've bound it to a different variable or the window is still open). Much of the power of Lisp derives from the ability to implicitly allocate and deallocate,

as well as to easily access, simple data structures like lists, or complex objects. In contrast, MacApp requires explicit method calls to allocate, initialize, and deallocate objects. In both cases, you must explicitly dispose of space that you've allocated from the Macintosh heap via Memory Manager calls. However, Common Lisp allocates space for its own objects and other data structures in its own heap area managed by the garbage collector.

We can compare the key features of ObjectPascal and Common Lisp object systems by inspecting the code needed to define two classes of objects, **`Beeper`** and **`LongBeeper.`** These classes have a **`BeepMe`** method that causes them to beep a number of times specified by an instance variable. **`LongBeeper`** inherits from **`Beeper. Beeper`** makes three short beeps, and **`LongBeeper`** makes four long beeps followed by the number of short beeps **`Beeper`** makes. In the ObjectPascal code, we abrogate specifications otherwise required by the ObjectPascal compiler that don't concern us.

Here's the ObjectPascal code:

```
TYPE
      TBeeper = OBJECT
            fBeeps:     integer;
            PROCEDURE TBeeper.IBeeper;
            PROCEDURE TBeeper.BeepMe;
            END;

      TLongBeeper = OBJECT(TBeeper)
            fLongBeeps: integer;
            PROCEDURE TLongBeeper.IBeeper;
            PROCEDURE TLongBeeper.BeepMe;
            END;

PROCEDURE TBeeper.IBeeper;
      BEGIN
            SELF.fBeeps := 3;
      END;

PROCEDURE TLongBeeper.IBeeper;
      BEGIN
            INHERITED IBeeper;
            SELF.fLongBeeps := 4;
      END;
```

```
PROCEDURE TBeeper.BeepMe;
      VAR   Count:      integer;
      BEGIN
            For Count := 1 to SELF.fBeeps do
                  SysBeep(30);
      END;

PROCEDURE TLongBeeper.BeepMe;
      VAR   Count:      integer;
      BEGIN
            For Count := 1 to SELF.fLongBeeps do
                  SysBeep(120);
            INHERITED BeepMe;
      END;

{A function that uses the LongBeeper class}

FUNCTION UseBeeper;
      VAR   myBeeper:   TLongBeeper;
      BEGIN
            NEW(myBeeper);
            FailNil(myBeeper);
            myBeeper.ILongBeeper;
            myBeeper.BeepMe;
            UseBeeper := myBeeper;
      END;
```

The same sequence in Common Lisp looks like this:

```
(defclass Beeper ()
      ((Beeps :initform 3)))

(defclass LongBeeper (Beeper)
      ((LongBeeps :initform 4)))

(defmethod BeepMe ((me Beeper))
      (dotimes (count (slot-value me 'Beeps))
            (_SysBeep :word 30)))

(defmethod BeepMe ((me LongBeeper))
      (dotimes (count (slot-value me 'LongBeeps))
            (_SysBeep :word 120))

      (call-next-method))

;;; A function that uses the LongBeeper
```

```
(defun UseBeeper ()
      (let ((myBeeper (make-instance 'LongBeeper)))
         (BeepMe myBeeper)
         myBeeper))
```

Although Common Lisp object system may at first sight seem to have more features than any particular programmer would need, in fact these capabilities are normally used by Lisp programmers.

Multiple inheritance is an instructive example. If you are trying to define classes with complementary behavior, multiple inheritance is the most elegant and economical solution. For example, you can define two classes called **ReadStream** and **WriteStream** that support read-only and write-only behavior for streams, respectively. This gives you the option of basing a class of read-write streams on inheritance from these classes:

```
(defclass ioStream (ReadStream WriteStream) ())
```

Since **ReadStream** and **WriteStream** are independent, you can also define a class of windows that act like write-only streams by inheriting from both the **Window** and **WriteStream** classes:

```
(defclass StreamWindow (Window WriteStream) ())
```

Using single inheritance to define **ioStream** and **Window** would result in redundant and unmodular code—one of the problems object-oriented programming tries to solve.

As you use multiple inheritance more seriously, however, you may have to deal with cases where you inherit multiple definitions of the same method. From the viewpoint of your class's semantics, you will probably want to do one of the following: (1) inherit all or some of the methods in any order or in a specific order, or (2) inherit none of the methods. Common Lisp allows you to deal with any of these possibilities. For example, to avoid inheriting a method, you simply redefine the method for the class you are defining without making a call to **call-next-method.** The latter is a generalization of ObjectPascal's **INHERITED** (compare above the **BeepMe** methods for the **LongBeeper** class in ObjectPascal and Common Lisp). Method combination is a feature that enables you to specify the order in which methods of a given name will be invoked.

"Before" and "after" methods enable you to specify behavior that should execute just before or after your method is invoked. This provides you with flexibility in method combination in subtle cases because the before and after methods are not embedded in the code of the primary method. But more interesting is the manner in which Common Lisp methods are dispatched. Whereas most object-oriented

systems dispatch on the class of the first argument, Common Lisp bases the method dispatch on the class of each argument passed to a method call. One example of a case in which you may want to dispatch on two arguments is when you have a **Print** method that can print on a variety of media. If you have a class **Document** that you want to be able to print into a **ColorLaser** stream or into an **ImageWriter** stream, you can define **Print** as follows:

```
(defmethod Print ((thingToPrint Document) (stream ColorLaser))
           ;; Code to print to a ColorLaser goes here
           )

(defmethod Print ((thingToPrint Document) (stream
ImageWriter))
           ;; Code to print to an ImageWriter goes here
           )
```

This generalizes object-oriented programming's idea that you shouldn't have to special-case your methods: the appropriate method will be called by the system on the basis of the type of *all* passed arguments. In particular, if the second argument (stream) is a ColorLaser, then the first method above will be called; if the stream is an ImageWriter, then the second method will be called. The alternative would be to check what kind of stream you are writing to within a monolithic **Print** method.

CLASS LIBRARIES AND EVENT SYSTEMS

Class libraries, which are provided with the language (some third parties sell alternative libraries or extensions), impose a model of how Macintosh events are handled, what kinds of Macintosh components (such as menus, dialog boxes) are available, and how these interact.

Both MacApp and MCL offer a set of classes to easily instantiate menu bars, menus, menu items, pull-down and pop-up menus, windows, dialogs, buttons, check boxes, static and editable text, lists, and spreadsheet tables. MacApp features extensive printer (via the **TPrintHandler** class) and undo (via the **TCommand** class) support. MCL features easy installation and handling of view objects, including menus and menu items.

Both systems support views as well as dialog and regular window classes. A view is an abstract way of defining rectangular drawing areas on the screen that are hierarchically related to other views. User interface objects, such as windows and buttons, inherit from the **view** class to get their scrolling behavior as well as their own coordinate system's origin. Scrolling a view scrolls its subviews within it, while the coordinate system of a view has its origin relative to the origin of its superview's coordinate system. Most useful toolbox controls and dialog items are predefined in both systems and are integrated with the view system. A set of event-handling methods defined on all views (for example, **activate**, **draw)** are automatically called by the event system, as necessary; the user need not be involved with the Macintosh inner event loop.

Table 2 compares the most important classes in MacApp and MCL.

You'll see examples of the use of some of these MCL classes in the later section "Now for an Example." For now, a note about the event system.

The MacApp class **`TApplication`** enables you to modify the handling of the Macintosh inner event loop. Instead of providing a comparable class, MCL enables you to optionally specify a function to which all events are passed. Your function can take any course of action; it can also override the regular MCL event-handling mechanism.

Table 2
A Comparison of Key Classes in MacApp and MCL

MacApp Class	MCL Class
TObject	Standard-Class
TAssociation	association list
TCommand	not applicable
TList	list[1]
TApplication	not applicable
TPrint Handler	not available
TDocument	not available
TView view	
TWindow	window
TDialogView	dialog
TTEView	dialog
TDialogTEView	dialog
TScroller	scroller-dialog-item[2]
TDeskScrapView	scrap-handler
TGridView	table-dialog-item
TTextGridView	table-dialog-item
TTextListView	sequence-dialog-item
TListView	sequence-dialog-item[3]
TClassListView	sequence-dialog-item[3]
TObjectView	sequence-dialog-item[3]
TControl	control-dialog-item
TCluster	view
TStaticText	static-text-dialog-item
TEditText	editable-text-dialog-item
TNumberText	editable-text-dialog-item
TIcon	icon-dialog-item
TPattern	not available
TPopup	pop-up-dialog[2]
TPicture	pict-dialog-item[2]
TCtlMgr	control-dialog-item
TButton	button-dialog-item
TRadio	radio-button-dialog-item
TCheckBox	check-box-dialog-item
TScrollBar	scroll-bar-dialog-item
TStream	stream
TFile	pathname
not available	fred-window
not available	menu
not available	menu-item

Notes:

1. MacApp includes a variety of list classes: these are required because of static language constraints. The regular Lisp list covers these cases.
2. These classes are distributed in example files with MCL.
3. These MacApp variations are due to static language constraints: they are handled within the sequence-dialog-item.

Macintosh OS events are regularly dispatched by MCL, even between the invocation of Lisp functions. Every few ticks (the number may vary with the version of MCL that you are using, but it usually is five ticks), MCL checks the event queue and if there's an event (including a null one), interprets the event and takes the appropriate action (for example, sends a **mouseDown** event to a view). Your functions can act as if they have full control of the Macintosh, since the event handling is opaque to the user. However, a macro called **without-interrupts** helps you to protect critical code segments, such as an operation on a Mac heap data structure (for instance, a window record) that might be disposed of by the code dispatched by an interrupting event (such as a click on the window's close box).

On the other hand, MacApp's **TView** class inherits its event-handling capability from the **TEvtHandler** class. For mouse event handling, since views are nested within each other, the most specific affected view will receive the mouse event: for example, the **HandleMouseDown** method of the most specific view under the mouse would be invoked on that view when the **mouseDown** event takes place. Events are processed by MacApp methods (primarily for **TApplication**, **TWindow**, **TDocument**, and certain view classes) until completion, blocking any other event handling.

In MacApp, one can generate **TCommand** objects, which can be created by menu or keyboard events. These objects will not only have methods that handle the event, but also conveniently store state information necessary to support **Redo** and **Undo**, which are necessarily associated with the event they represent. This provides a nice framework for **Undo** support, at the cost of generating and maintaining these objects. In addition, event chains can be specified within MacApp: this enables you to specify a chain of event-handling objects that are candidates for handling specific types of events. MacApp will cycle through the chain to find an object that can handle that event and invoke the appropriate method on that object. Similarly, "target chains" allow you to specify hierarchies of objects that are candidates for handling events.

In general, the MacApp event-handling system is far more articulated than MCL's. The cost of this articulation is increased complexity and some reduced flexibility. Once you understand the MacApp event system and find its constraints acceptable for your application, you may find that your workload is reduced.

Note also that menus and menu items are handled in MacApp via special menu resources. At run time, MacApp invokes the **DoMenuCommand** method on the **TApplication** instance; the latter must interpret what to do on the basis of the chosen menu item.

STRONG POINTS OF THE DEVELOPMENT ENVIRONMENTS

Both MacApp (in conjunction with MPW) and MCL offer excellent development environments. Still, each has its strong points, which we'll focus on here.

The primary advantage of the MCL environment is the ability it gives you to incrementally compile your code: you can recompile one function (or even a single expression or statement) at a time. This fact has far-reaching implications for how you develop your program.

Once you compile a function, it is automatically linked to the rest of your code. This means that you can immediately test your function. If it does not work, you either (1) go to the source code of another function and change it, recompile it, and retry your original function, or (2) modify, recompile, and retry your function. The MCL (and every other Lisp-based) debugging and browsing environment is geared to this kind of activity. Jumping from one function's source text to another one, perhaps in different files in different directories or volumes, is a simple matter. And you are not required to execute your program from a "main" but can try out each function as a separate module.

MCL provides you with the following powerful debugging tools:

- Inspector—Enables you to inspect any value of any instance variable of any instance and also to change that value at any time. You can inspect any Lisp structure, class, or class instance. For functions or methods, the Inspector will provide you with disassembled code. Information about devices, files, packages, and other system objects can be conveniently obtained via the Inspector.
- Backtracer—Enables you to examine the execution state of your program by looking at the program as a set of frames (like a movie's frames). Each frame is usually a function or method invocation, with information about the state of all local variables and objects last referenced. You can move through these frames backward or forward to the point where the error or user break occurred and can change any state in a frame.
- List of definitions—Provides you with a list of definitions of variables, functions, methods, and macros within your text files. You can go to any specific entry by double-clicking on it.
- Apropos—Allows you to search for information about any object in the system based on its name.
- Error handling—Gives the ability to signal errors and general conditions. Dead-end and continuable errors are supported. Continuable errors enable the user to change the environment so that the error state is reset and processing can restart.

See the sidebar "Evaluating and Navigating the Code in MCL" for key details about the MCL development environment.

In contrast, although in MacApp you can separately compile a class definition (an ObjectPascal Unit file), to actually test it you must link it to the rest of your program. Debugging support must be explicitly requested from the compiler and linker; debugging code is embedded in your running code. Fortunately, the MacApp browser provides an excellent browsing facility, though it can only read, not write, due to the absence of incremental compilation. MacApp does provide a debug transcript window and a debug menu, and Inspector windows when you've compiled your MacApp program with debug and inspect code, respectively. The Inspector window enables you to examine the values of fields of class instances. To support the Inspector window you must write methods for each class you define that know how to supply the necessary information—that is, field names and values for any instance—to the Inspector.

MacApp and MCL both provide tools to directly manipulate the user interface: ViewEdit™ and the MCL Interface Tools (IFT), respectively. ViewEdit works mostly by allowing you to create views and put objects into the views, as necessary: the result is a set of resources that can be saved for your MacApp application. IFT currently does not let you define views, but only dialogs and menus. You can create new and modify existing menu, menu item, dialog, and dialog item objects: the result is actual Lisp code that can regenerate those objects.

The reason ViewEdit generates resources is that the architecture of some parts of MacApp rely on the coordination with existing resources (for example, menus), whereas IFT considers resources as optional additions. IFT can be used for instructional purposes: for instance, just to see how to program the user interface objects you create via direct manipulation. Since IFT does not, like Prototyper™, build a complete application for you, one tends to use IFT to assist the creation of user interface code rather than as a replacement for doing the main work.

Although both MCL and MacApp/MPW encourage the use of existing class libraries rather than low-level system access, substantial applications invariably require direct access either to build new kinds of objects or to increase overall performance. The MCL toolbox interface can be accessed from two levels: (1) as regular low-level toolbox register or stack traps, and (2) through predefined higher-level methods available with the class library (for example, a QuickDraw library).

EVALUATING AND NAVIGATING THE CODE IN MCL

When you first open an MCL application, you see the window known as the Lisp Listener. The Listener serves the same purpose as the message box does for HyperCard. It reads whatever you type into it, evaluates it as a Lisp expression, and prints out the value returned by the evaluation (remember that *all* Lisp expressions return a value). This process is called the read-eval-print loop.

When you evaluate expressions in the Listener or in text windows (which you get by choosing New from the File menu), you get immediate feedback without having to explicitly compile or link the expressions you enter. Similarly, evaluating a function you've just coded means that the function is already compiled and linked: you are ready to use it!

Navigating the code in MCL is also a simple matter. If you are looking for the source of a function but are not sure what it is called, you can use the Apropos dialog. You suggest name fragments, and get back the names of variables, functions, methods, or other defined symbols that contain the fragment. If you find what you're looking for there, you need only double-click on the returned name for the system to bring up the file containing the desired definition and to position the cursor at the beginning of it.

When you are studying the code of the function and you see functions it calls that you want to inspect, you need only click on the function name with the Command and Option keys pressed, and its definition will be shown to you in a text window.

If you want to optimize your function, you can inspect it by using the Inspector—for example, **`(inspect`**

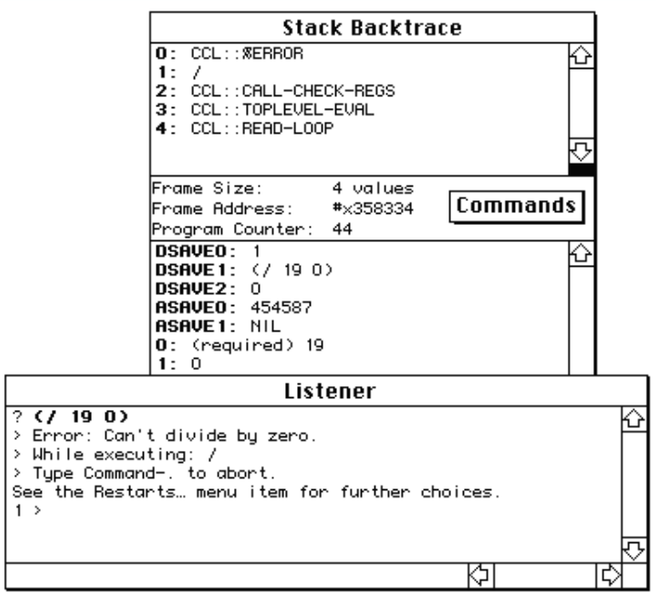


Figure 1
A Sample Backtrace

(fboundp 'my-function-name). The Inspector window includes the disassembled code of the function. Or you can more directly examine its assembler code by using the disassembler—for example, **(disassemble 'my-function-name)**.

When an error is signaled, you can use a Backtrace window to check how you got there—for example, to see which functions were called before this one, which parameters were assigned to them, and to inspect the values of local variables. (Figure 1 shows a backtrace for a division by zero error.) You can automatically invoke the Inspector on any value in the Backtrace window by double-clicking on that value.

If you want to get documentation for a function, you can click on the name of the function and choose the Documentation menu item, or press Control-X followed by Control-D to get it. If you'd like to get the formal argument list for it, you can click on its name and press Control-X followed by Control-A.

Once you're inside a text window (officially called a Fred* window), you can do incremental searches backward and forward without the need of dialogs. In addition, you can easily skip around nested expressions, transpose expressions, words, or characters, and skip forward or backward through definitions. A window containing a list of the definitions in the Fred window can be selected from the menu: this allows you to directly go to a definition by double-clicking on the definition's name. (Figure 2 shows a Fred window accompanied by a window with list of definitions.) And if you can't find a way to do something, you can always extend the programmable Fred Editor . . . and send the extension to me!

* Fred is an acronym for "Fred resembles Emacs deliberately."

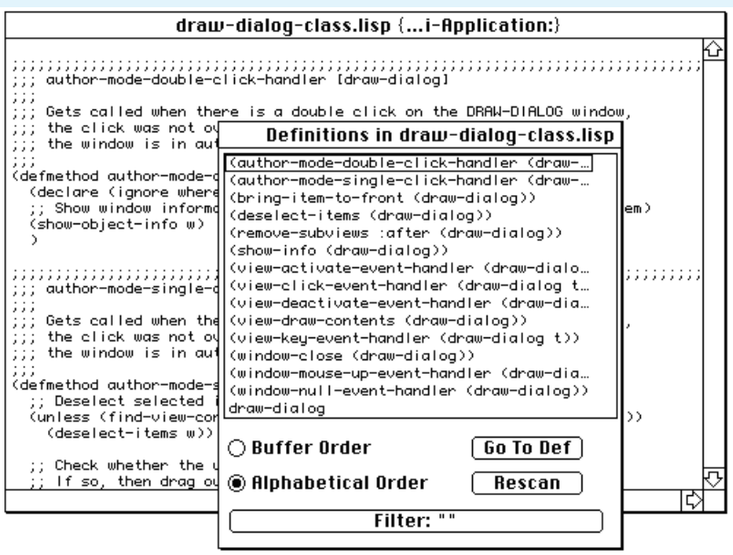


Figure 2
A Sample Fred Window

Pascal-style record definitions, with variant capability like PL/I's, can be used to define any toolbox handle or pointer-based records. For example, a rect would be defined like this:

```
(defrecord Rect
  (variant ((top integer 0)
            (left integer 0))
           ((topleft point)))
  (variant ((bottom integer 0)
            (right integer 0))
           ((bottomright point))))
```

This variant specification enables us to describe the rect in any reasonable combination of top, left, bottom, right, top-left, or bottom-right points. We use the following code to create a rect and bind it to the variable **myRect:**

```
(setq myRect (make-record :rect :top 0 :left 0
                                :bottom 100
                                :right 100))
```

This calls the Macintosh Memory Manager. A call to the toolbox trap **invalrect** would look like this:

```
(_InvalRect :ptr myRect)
```

The only argument to this stack trap is the pointer to the rect.

Similarly, to allocate a handle:

```
(_NewHandle :check-error :d0 80 :a0)
```

The **:check-error** parameter tells MCL to signal an error, with error information, if the trap returns an error. Here, **:d0** and **:a0** specify the 68000 registers to be used in this register trap. 80 bytes are requested in **:d0**, whereas the trap call is asked to return the value in **:a0** (that is, for **NewHandle**, this is the address of the handle).

The MCL foreign function interface will read any MPW-format object file and load any specific or all entrypoints into MCL. Then, MCL will link the loaded procedures with the required MPW libraries. From the viewpoint of MCL, the foreign function is accessed as if it were a Lisp function. For example, if we've compiled a C function called **StringToNumber** into the file myProgram.o, we can load and link its code by evaluating the following form:

```
(ff-load "ff;myProgram.o"
         :ffenv-name 'sample-of-linking-to-mpw
         :libraries '("clib;StdCLib.o"
                      "clib;CInterface.o"
                      "mpwlib;interface.o"))
```

Now, to define a Lisp function that acts as if it were the C function, we evaluate the following definition:

```
(deffcfun (StrToNumb "StringToNumber") (string)
      :long)
```

This creates a Lisp function called **StrToNumb** that takes a string as its input and returns a long integer. Hereafter, we can call it as follows:

```
      (StrToNumb "198")
```

SIZE AND PERFORMANCE SPECIFICATIONS

The price of MCL's flexibility and ease of use becomes apparent when we compare its size and performance specifications to those of MacApp. While a small MacApp program might require as little as 40 KB, at press time, MCL 2.0 requires a minimum of more than 2MB. (Please consult APDA for specific information on the size requirements for the released product.) A long-term goal of MCL is to reduce this size by not loading functions you don't use.

Insofar as speed is concerned, at the time of this writing the method lookup for MCL is 16 microseconds on a Mac IIcx or 8 microseconds on a Mac IIfx when dispatching on one argument. The method dispatch in MCL depends somewhat on the number of a method's arguments, whereas MacApp dispatches on the class of the object only. But while the speed of MCL is certainly inferior to that of compiled Pascal, with faster machines and better compiler technology speed is no longer critical for real applications.

The other major disadvantage of MCL is the ubiquitous interference of the garbage collector. The mark-and-copy garbage collector can freeze the system for up to ten seconds in its current implementation. This can be inconvenient, to say the least, to your application's user. Fortunately, work is under way to provide an ephemeral garbage collector that will incrementally work in the background under the virtual memory feature of System 7.0. One way to postpone garbage collection (and, incidentally, often to speed up your code) is to minimize consing—that is, to write code that uses arrays instead of the commonplace list structures, or, better yet, code that uses its own memory management by keeping a private pool of free objects. Of course, the advantages of garbage collection (for example, no memory leaks, no need to worry about allocating and deallocating memory) should not be forgotten. Since

vectors and lists can be treated as **Sequence** data types, it is possible to write code that can deal with either data structure.

NOW FOR AN EXAMPLE

You'll find an example of a program developed in MCL in the LISP folder on the *Developer Essentials* disc that accompanies this issue. This sample application combines some of the features of programs like MacDraw® and HyperCard. Its main components are as follows:

- Windows in which the user can build graphics
- Palettes from which tools can be selected and from which graphic objects can be dragged out
- A variety of graphic objects (for example, buttons, fields) that can be dragged from the palettes to build something in the windows
- Menus to simplify the user interface
- An event system that approximates HyperCard application's, including the ability to write scripts for objects—albeit in Lisp

In the same folder, you'll also find "All About the MiniLisp Application," an article that takes you step by step through the development of the application, in case you really want to learn the details of how to program using MCL.

In the remainder of this article, I'll show selected fragments from that example program to illustrate some of the features and advantages of MCL mentioned earlier.

BUILDING A MENU

In our sample application, our File menu has New, Close, and Quit menu items. We start building the File menu like this:

```
(defvar *mini-application-file-menu*
      (make-instance 'menu :menu-title "File"))
```

First, **defvar** is the Lisp function we use to define and initialize a global variable: we have thus initialized the variable ***mini-application-file-menu*** to contain the menu instance. Note that Lisp identifiers can be of any length and can consist of any character; by convention, names of global variables start and end with an asterisk. The method **make-instance,** which is similar to **New** in Smalltalk and in MacApp, enables us to create an instance from a class, and to optionally initialize some of its slots. (The term *slot* is synonymous with *instance variable* in other object-oriented languages.) The first argument to **make-instance** is the name of the class, and the subsequent arguments are keyword-value pairs with optional initializations that depend on the class. The predefined MCL class **menu** is instantiated and the **:menu-title** keyword option is initialized with the name

of the menu. (The name **menu** is preceded with a single quotation mark to indicate that we are referring to the symbol **menu** rather than to the variable *menu*.)

To test this instance, we invoke the **menu-install** method on it:

```
(menu-install *mini-application-file-menu*)
```

The menu bar now includes a new File menu. To define a menu item, we evaluate the following:

```
(defvar *file-new-menu-item*
        (make-instance 'menu-item
                :menu-item-title "New"
                :command-key #\N
                :menu-item-action 'new-menu-item-action))
```

The predefined MCL class **menu-item** supports, among other things, an optional command-key character (characters in Lisp are represented by prefixing the character with #\) and a menu item action. The latter is a function that will be called when the menu item is selected. In this example, we define the keystroke combination **Command-N** to be equivalent to selecting the menu item. We have also given **new-menu-item-action** as the name of the function that should be called when this menu item is selected. The additional keywords **:disabled**, **:menu-item-color**, **:menu-item-checked**, and **:style** (none of which are used here) enable us to specify whether the menu item should be disabled, the color of the parts of the menu (such as background, text), whether the item should be checked, and its style.

We install the new menu item on the File menu by typing and evaluating the following in a text window:

```
(add-menu-items *mini-application-file-menu*
                *file-new-menu-item*)
```

add-menu-items is a predefined MCL method for menus that allows us to add one or more menu items to a menu—whether the menu is installed or not.

CUSTOMIZING OUR WINDOW

We customize the MCL window for our application by creating a subclass of the predefined **window** class. Our new class includes a slot **my-items** that will hold a list of all the graphic objects that we draw into the window, remembering their back-to-front ordering. We start by creating a subclass of the **window** class named **draw-dialog**. **defclass** is the CLOS function for defining a class:

```
(defclass draw-dialog (window)
```

```
((my-items :initarg my-items :initform NIL)          ; Items in window
 (item-last-under-mouse :initarg item-last-under-mouse
                        :initform NIL)               ; Item under the mouse
 (browse-mode :initarg browse-mode:initform NIL)     ; Window mode
 (selections :initarg selections :initform NIL))     ; Item(s) now selected
(:documentation "This class defines our windows"))
```

Our class **draw-dialog** adds four slots named **my-items**, **browse-mode**, **item-last-under-mouse**, and **selections** to its superclass, **window**. The first argument to **defclass** is the name of the class, **draw-dialog**, followed by a list with the names of all classes, if any, from which we want to inherit (that is, that we want our class to be a subclass of)—in our case, just the **window** class. As mentioned earlier, multiple inheritance is supported and although things are, by default, inherited in the order in which they appear in this list, protocols are available in the CLOS specification for controlling this. (This "meta-object" protocol is not yet supported in MCL.) The third and fourth arguments shown here are a list of descriptors for each new slot that we want to define and documentation text, respectively.

A descriptor is a list that starts with the name of the slot followed by a set of keyword-value pairs that represent options concerning how the slots get initialized when an instance of this class is created. We used the **:initform** keyword followed by the the expression NIL. This means that when an instance of **draw-dialog** is created, the **my-items** slot will by default be set to whatever the following expression evaluates to—that is, NIL. The **:initarg** keyword, on the other hand, is followed by the name by which this slot will be recognized in future slot accesses—in particular, how the slot will be referenced within a call to **make-instance.**

The **draw-dialog** class will inherit slots from the **window** class and from the classes it inherits from. We can verify that this new subclass definition works by creating an instance of it:

```
(setq my-window (make-instance 'draw-dialog))
```

One way to check whether the slot **my-items** has been added and initialized to NIL is to get its value directly, via the expression

```
(slot-value my-window 'my-items)
```

Another way to check this slot is by using the Lisp Inspector to view the object instance. The Inspector provides us with a description of any object (this includes constants, variables, and functions) that we want to look at. In addition the Inspector will allow us to directly edit values. To inspect the instance in **my-window**, we evaluate **(inspect my-window)**, which brings up a screen that looks like Figure 3.

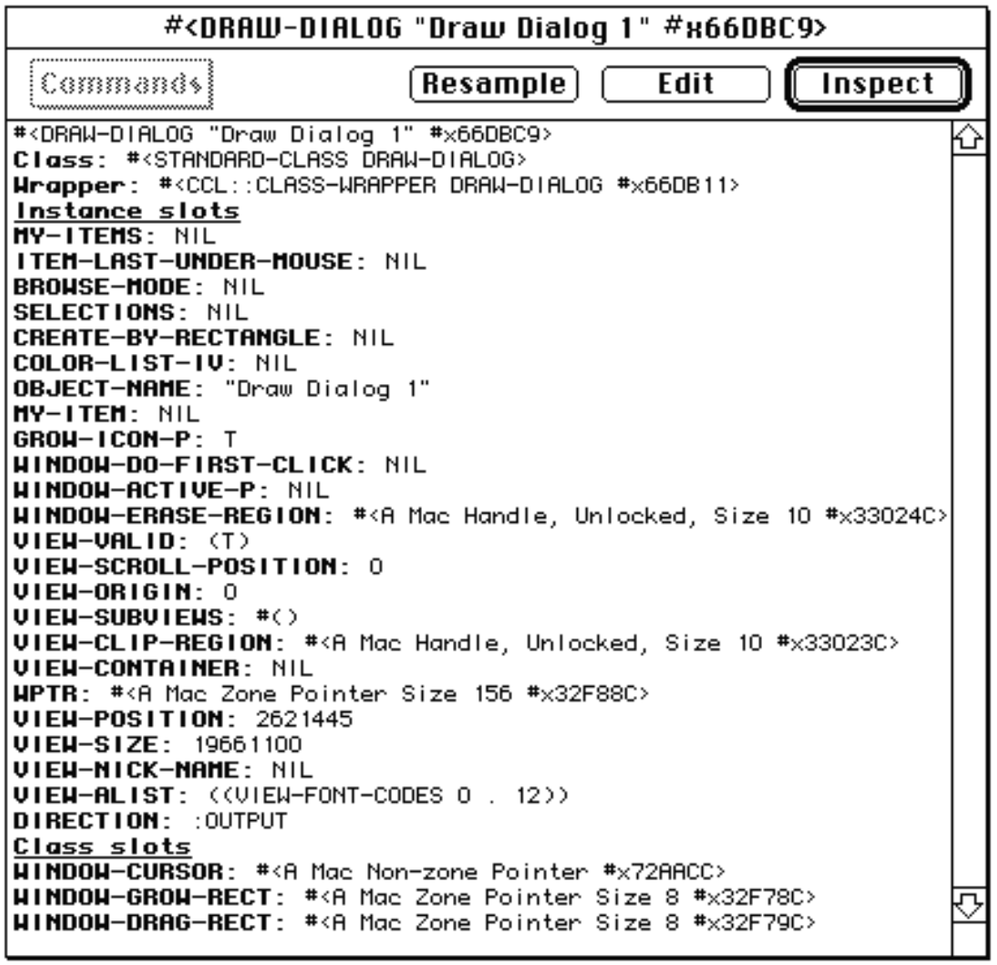


Figure 3
Viewing an Instance of the draw-dialog Class in the Inspector

The items following **`Local slots:`** are slots of the object bound to **`my-window`**. The slots that we added should be at the top, as shown; the remaining slots have been inherited from the **`window`** class. For example, the slot **`wptr`** contains the pointer to the Macintosh window definition block, and **`view-size`** is the point (that is, a long used as a point) for the window's size. To inspect the window block itself, we can double-click on the line in Figure 3 with the **`wptr`** to get the display shown in Figure 4.

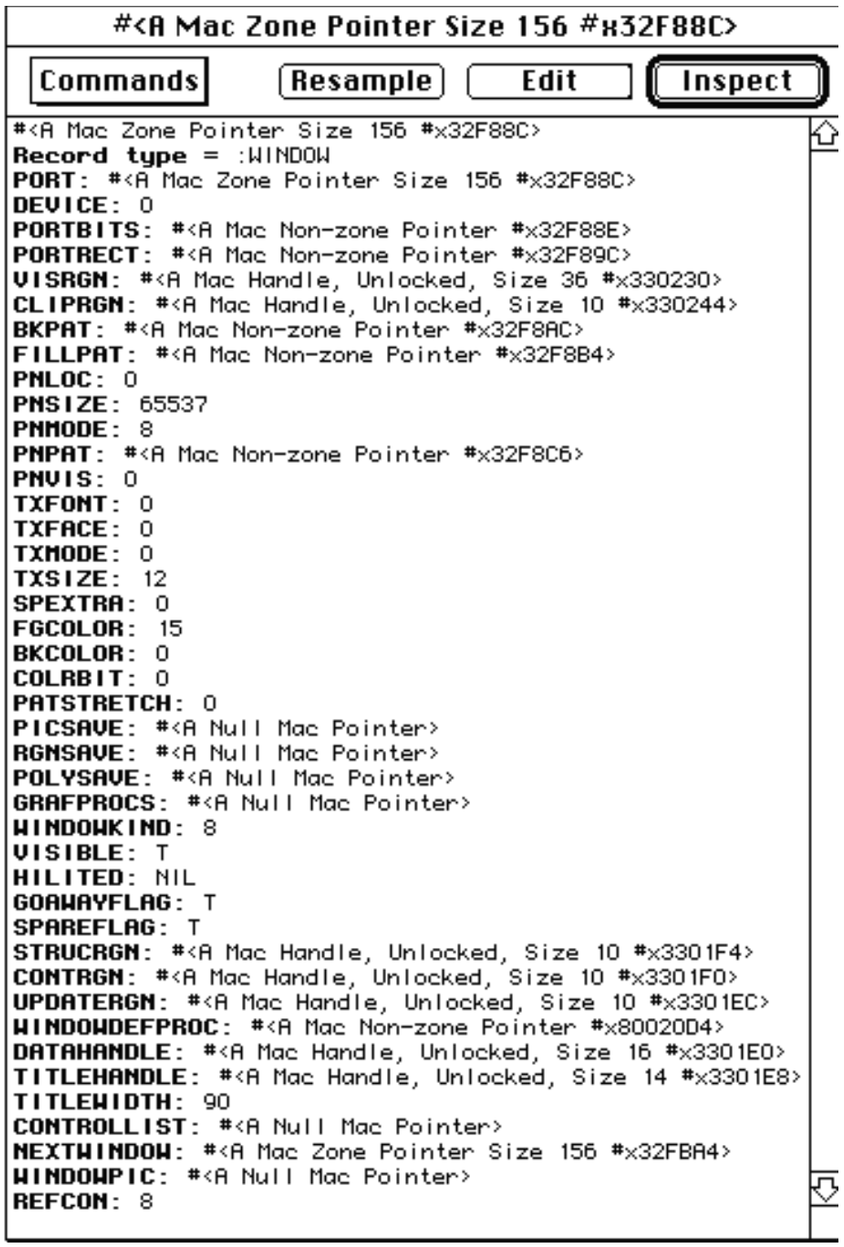


Figure 4
Viewing the Window Record for Our Instance of draw-dialog in the Inspector

We could continue this process indefinitely—for example, looking next at the rectangle records in the window record. This illustrates a point made earlier: although Lisp is a very high-level language, you can still access the system as if writing in assembler language. This clarifies toolbox access considerably when compared with C and Pascal.

The complete source code goes on to define the behavior of our window, including making the window handle events in the way that HyperCard does. See the CD-ROM if you are interested in details.

CREATING OUR PALETTE

Our palette has two kinds of objects in it:

- Tools to select what should be done
- Objects that can be dragged into our windows

The **draw-dialog** class can do most of the work for us, so we define a subclass of it called **palette** as follows:

```
(defclass palette (draw-dialog)
  ((my-tools :initarg :tools)
   (my-draw-items :initarg :draw-items))
  (:documentation "Palettes used in our application"))
```

The **my-tools** slot will contain all the items in the palette that can be viewed as tools, whereas the **my-draw-items** slot will contain all the items that can be dragged out of the palette into other windows. These slots will be initialized with instances of tools and graphic items that will be used in any one session of our application. Once a palette is created and initialized, a layout method (in CD ROM sources) will look at these slots to figure out how to lay out the items—for example, tools first and draggable items next. These are convenience slots, since graphic items themselves know whether they are tools or not.

We have to enforce the following differences between our **palette** and **draw-dialog** classes:

- Since tools are not accessible to users, a convenient place to put the code that does the tool's work when it is clicked is the tool's **mouse-down** event handler.
- Items in our palette cannot be moved within or resized in the palette.
- Tools cannot be dragged out of the palette.

First, we want to make sure that a palette's tools get the **mouse-down** event whether or not we are in author mode. We redefine **view-click-event-handler** this way:

```
(defmethod view-click-event-handler ((palette palette)
where)
   (let ((item (find-view-containing-point palette where)))
      (if (slot-value item 'tool)
         (mouse-down item where))   ; dispatch the
                                    ; mouse-down event
      (call-next-method)))          ; proceed with the
                                    ; usual behavior
```

If an item is selected and if it is a tool, we force the **mouse-down** and then call the **draw-dialog**'s **view-click-event-handler** method using **8call-next-method.** The check **(slot-value item 'tool)** anticipates that the tool slot of an item tells it whether it is a tool or not.

Finally, since the resizing and dragging is done by the items themselves, items that know themselves to be in the palette should not allow themselves to be dragged or resized around a palette (but they should let themselves be dragged to other windows!). Similarly, items that are tools will know better than to drag themselves out of a palette!

CREATING DRAW ITEMS

In our application, the graphic items have been delegated most of the work by the other classes. We define one basic class of graphic items from which tools and other kinds of user interface objects will derive.

```
(defclass draw-item (dialog-item)
  ((rectangle :initarg :rect :initform nil)
   (tool :initarg :tool :initform nil)
   (selected :initform nil)
   (name :initarg :name :initform ""))
  (:documentation "The user interface objects"))
```

We call our class **draw-item.** It will be a subclass of MCL's **dialog-item** class. The latter class supports a variety of specializations for typical Macintosh user interface items, like radio and round buttons, check boxes, and static text. Since we don't want to duplicate that functionality, we subclass from **dialog-item.** Since **dialog-item** itself inherits from the class **view** (actually, from **simple-view**—but let's not split hairs here), we get all the functionality we need without using multiple inheritance! We can verify this provenance by inspecting the class object for **draw-item** using the Inspector, as shown in Figure 5.

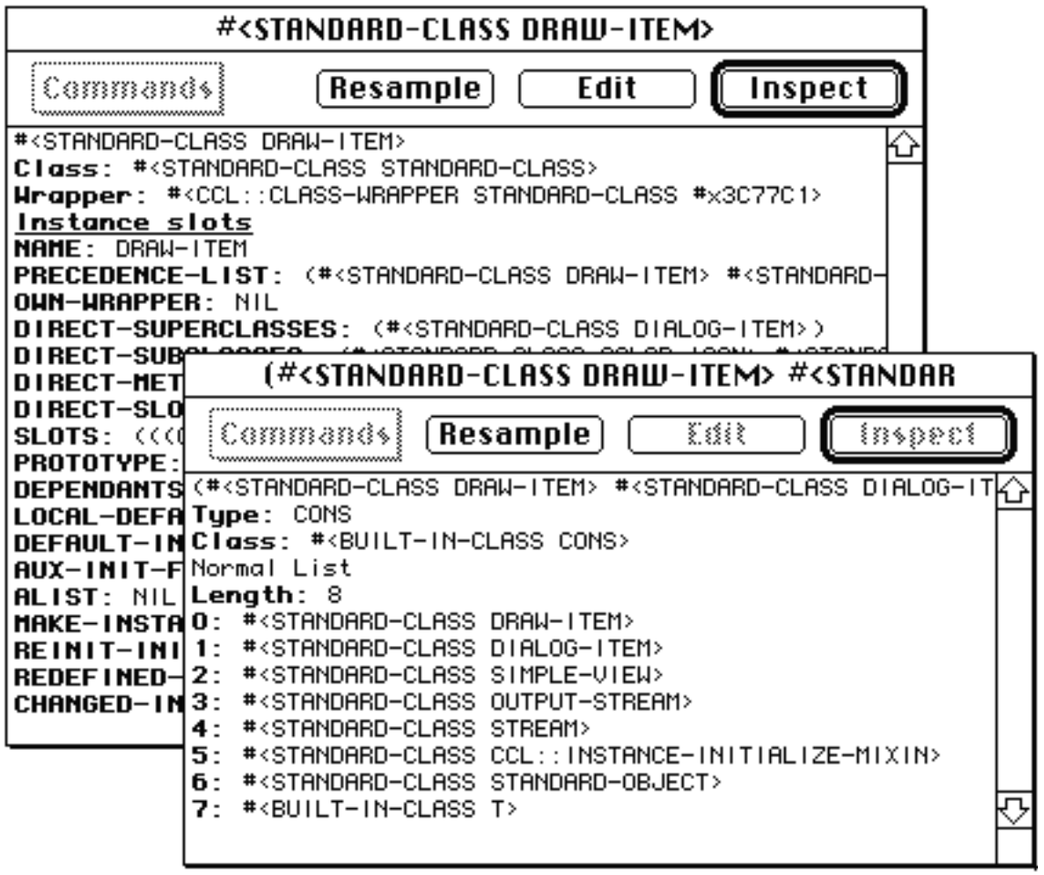


Figure 5
The Class Precedence List for draw-item

You will notice that our **draw-item** not only inherits from **simple-view**, but also from **stream.** The latter class allows one to apply **format**, **print**, and other **stream** (input/output) methods to our items!

draw-item's **rectangle** slot will be used to keep the bounds of the **draw-item.** These bounds will be the same as one would obtain by using the methods **view-position** and **view-size** when applied to the item, but will be much more efficient than making two method calls. The **tool** slot tells us whether the item is a tool or not. The **selected** slot tells us whether the item is

selected (note that this information is redundant since it is also maintained in the window s **selections** slot). Finally, a **name** for the draw item.

Our code goes on to define the behavior of the draw item. We won t take the space to discuss it here, but do want to show the **resize** and **drag** methods.

The **resize** method looks like this:

```
(defmethod resize ((item draw-item) current-mouse-loc)
         (let* ((resize-direction (get-resize-direction
                                    item current-mouse-loc))
         (topleft (view-position item))
         (size (view-size item))
         (bottomright (add-points topleft size))
         (top (point-v topleft))
         (left (point-h topleft))
         (bottom (point-v bottomright))
         (right (point-h bottomright))
         new-mouse-loc new-mouse-h new-mouse-v
         ;; Two regions to produce inverted effect:
         (old-resize-region (new-region))
         (new-resize-region (new-region))
         ;; The rectangle enclosing the window
         (window-rectangle (rref (wptr item)
      :window.portrect))
         ;; The rectangle enclosing the draw-item:
         (item-rectangle (slot-value item 'rectangle))
         (window (view-container item)))
    (_inverrect :ptr item-rectangle)
    (unwind-protect
      (loop ; until the mouse is released
        (if (not (mouse-down-p))
          (return nil)) ; We're through!
  ;; Update the location of the mouse in window coordinates
        (setq new-mouse-loc (view-mouse-position window)
              new-mouse-h (point-h new-mouse-loc)
              new-mouse-v (point-v new-mouse-loc))
        ;; Do resize graphics if mouse is within the window:
        (when (point-in-rect-p window-rectangle new-mouse-loc)
          (_RectRgn :ptr old-resize-region
                    :ptr item-rectangle)
          (case resize-direction
            (:top (and (< new-mouse-v bottom)
                       (rset item-rectangle
                             :rect.top new-mouse-v)))
```

```
                (:bottom (and (> new-mouse-v top)
                              (rset item-rectangle
                                    :rect.bottom new-mouse-v)))
                (:left (and (< new-mouse-h right)
                            (rset item-rectangle
                                  :rect.left new-mouse-h)))
                (:right (and (> new-mouse-h left)
                             (rset item-rectangle
                                   :rect.right new-mouse-h)))
                (:topleft (and (< new-mouse-v bottom)
                               (< new-mouse-h right)
                               (rset item-rectangle
                                     :rect.topleft new-mouse-loc)))
                (:topright (when (and (< new-mouse-v bottom)
                                      (> new-mouse-h left))
                             (rset item-rectangle
                                   :rect.right new-mouse-h)
                             (rset item-rectangle
                                   :rect.top new-mouse-v)))
                (:bottomleft (when (and (> new-mouse-v top)
                                        (< new-mouse-h right))
                               (rset item-rectangle
                                     :rect.left new-mouse-h)
                               (rset item-rectangle
                                     :rect.bottom new-mouse-v)))
                (:bottomright (and (> new-mouse-v top)
                                   (> new-mouse-h left)
                                   (rset item-rectangle
                                         :rect.bottomright new-mouse-loc))))
          (_RectRgn :ptr new-resize-region :ptr item-rectangle)
          (_xorrgn :ptr new-resize-region :ptr old-resize-region
                   :ptr old-resize-region)
              (_inverRgn :ptr old-resize-region)
              ))
           (_inverrect :ptr item-rectangle)
           (set-view-size item (subtract-points
                                          (rref item-rectangle
                                          :rect.bottomright)
                                (rref item-rectangle
                                                :rect.topleft)))
      (set-view-position item (rref item-rectangle :rect.topleft))
      (dispose-region old-resize-region)
      (dispose-region new-resize-region))))
```

The **resize** method illustrates the use of direct stack-based toolbox calls: these are the functions whose names are prefixed by the underscore (_) character, like **_inverrect**. In these calls, you can only pass a pointer, a single word, or a double word. You must take all the precautions you would if you were calling the traps from assembler. As a matter of fact, you *are* at assembler level when you make these calls: you can pass pointers or handles or a long number in a double word: you re the boss.

Another important feature of Common Lisp to observe in **resize** is the **unwind-protect** construct. This enables you to protect an expression (in this case, the long expression enclosed within the **loop)** in case there s an error during its execution. The **unwind-protect** guarantees that the expressions following the protected expression will be executed despite the error. We thus ensure that all the regions created for **resize** will be disposed of even if the routine crashes.

Here s the **drag** method:

```
(defmethod drag ((item draw-item) current-mouse-loc)
  (let ((start-position (view-position item))   ; Start of the drag
        (end-position nil)                      ; End of drag
        (item-region (new-region))              ; What are we dragging?
        (window (view-container item))          ; Where are we dragging?
        (destination-window nil)                ; Where did drag end?
        (drag-offset nil))                      ; Drag offset

    (unwind-protect ; dispose regions despite errors
      (progn

        ;; Define the region that we want to drag:
        (open-region window)
        (with-port (wptr item)
          (_framerect :ptr (slot-value item 'rectangle)))
        (close-region window item-region)

        ;; Do the drag and get the offset of the drag:
        (setq drag-offset
              (drag-inverted-region (view-container item)
                  item-region :start current-mouse-loc))

        ;; Find out in which window the item landed:
        (setq end-position (add-points start-position drag-offset)
              destination-window (find-draw-dialog-in-point
                                  (add-points end-position
                                              (view-position window))))
```

```
  (when destination-window    ; Do nothing if it lands nowhere
    (if (eq window destination-window)
      ;; Move within this window:  set the item's
      ;; position at the end of the drag:
      (unless (eq (type-of window) 'PALETTE)   ; No drags within PALETTE
        (set-view-position item end-position))
      ; Move to another window: drop it there
      (move-item-to-window item end-position window destination-window))
    (view-draw-contents item)))

(dispose-region item-region))))
```

The **drag** method creates a region around the bounds of the thing to be dragged and calls the function **drag-inverted-region**, which enables drags to occur between as well as within windows and returns an offset to where the item was dragged. If the destination window is the same as the window where the drag started and the window is a palette, we want to do nothing since we can t have a drag within a palette. Otherwise, we adjust the position of the item using **set-view-position**. If the move is to another window, then we use **move-item-to-window**. The latter will interpret a drag as a clone of the item if the drag started on a palette and ended in a **draw-dialog** window, or else it will interpret it as a simple drag. **move-item-to-window** uses the MCL **view** methods **add-subviews** and **remove-subviews** to add and remove the items from the source and destination windows. If there s a clone, the function **clone-draw-item** is called and the item is added to the destination window. **move-item-to-window** ends by selecting the destination window.

CREATING A DOUBLE-CLICKABLE APPLICATION

You can create a double-clickable version of your application using the MCL function **save-application**. This allows you to distribute your final application for run-time use, without the rest of the MCL development environment.

TO SUM UP

Macintosh Common Lisp offers many advantages that are beginning to look more attractive to developers as faster machines and cheaper memory minimize its drawbacks. Advantages like multiple inheritance; typeless variables; abstraction away from Macintosh event-loop style of programming; the ability to implicitly allocate and deallocate, as well as to easily access, simple data structures like lists or complex objects; a large and consistent class library; and the ability to incrementally compile code outweigh MCL slower speed and larger memory requirements. If this taste of MCL has made you want to learn more, see the *Developer Essentials* disc and the following list of recommended reading.

RECOMMENDED READING

REFERENCES

Common Lisp: The Language, 2nd ed., by Guy Steele (Digital Equipment Corporation, 1990).
This is the language book of last resort not only for programmers but also for implementors of the Common Lisp standard. A must for any serious programmer, it is nevertheless too lengthy and detailed to serve as a quick reference guide: at 1,029 pages, it makes the Old Testament look inviting. The second edition contains changes and extensions to the standard, including the addition of the Common Lisp Object System (CLOS).

Common Lisp: The Reference by Franz, Inc. (Addison-Wesley, 1988).
This is a useful alphabetically ordered reference book on Common Lisp. Unfortunately, at the time of this writing there is no revised version of it that matches Steele's second edition.

Object-Oriented Programming in Common Lisp: A Programmer's Guide to CLOS by Sonya E. Keene (Addison-Wesley, 1989).
After you've waded through the 1,029 pages of Steele's book, this will quickly land you on Mother Earth. This book does much to disentangle the complexity of the Common Lisp Object System. Keene's book will be a companion to your Common Lisp reference.

Macintosh Common Lisp 2.0 Reference (Apple Computer, 1990).
This is Apple's reference for MCL. It describes all user interface classes, the event system, and the development environment, and gives you everything you need to get started—except for the application.

TUTORIALS

Lisp tutorial books vary in quality. There is no substitute for going to a well-stocked technical bookstore and taking the time to select the book that appeals most to you. However, you should not miss the following three:

Lisp, 3rd ed., by P. H. Winston and B. K. P. Horn (Addison-Wesley, 1989).
If you enjoyed college coursework , you'll love this book. It is replete with problem sets and includes an instructive and interesting variety of examples. The essentials of the Lisp language and of Lisp thinking are covered quite nicely—if you have the stamina to systematically work your way through it all.

Common LISPcraft by Robert Wilensky (Norton, 1986).
Ranking in serviceability midway between the other two tutorial books listed here, this one manages to avoid the tedium of the one and the shallowness of the other. Its examples are short and direct. It contains lucid explanations of some subtle issues (for example, funargs and binding) that in other contexts could appear alien to you.

LISP: A Gentle Introduction to Symbolic Computation by David S. Touretzky (Harper & Row, 1984).
If you are intimidated by Lisp, this may be the book for you. Touretzky takes great pains to make sure that you understand every point he sets out to make—particularly about the fundamentals of the language (for example, "what is a list?"). The only problem with this is that the only place to go from here is to another, more thorough tutorial book. But we all must begin somewhere!

Thanks to Our Technical Reviewers
Yu-Ying Chow, Bill St. Clair, Sarah Smith, Jim Spohrer, Steve Weyer •

INDEX

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

E

F

G

H

I

INDEX

INDEX

T

U

V

W

X, Y

Z