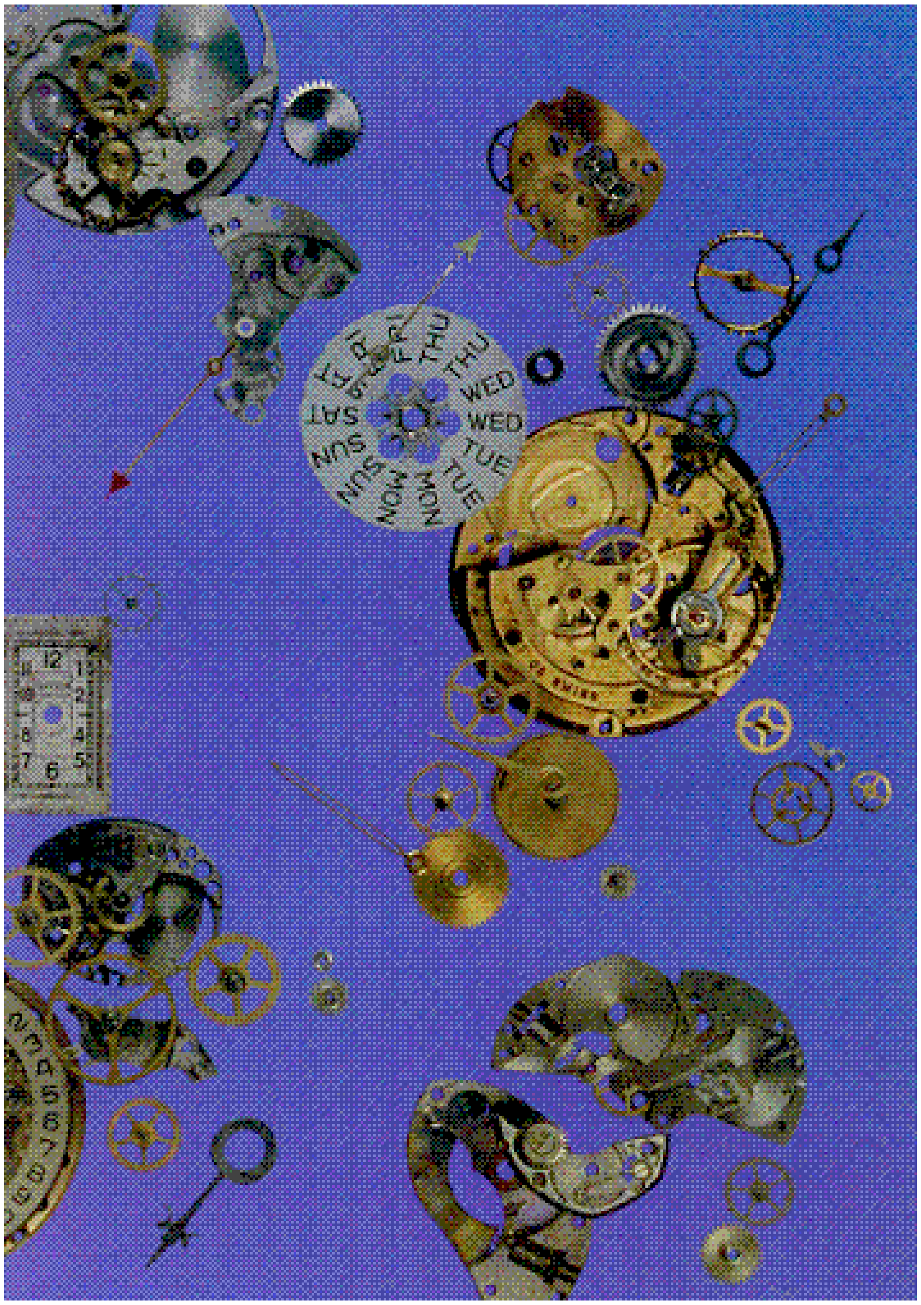
develop

The Apple Technical Journal



Issue 4 October 1990

Apple Computer, Inc.

E D I T O R I A L

Editor in Chief's Clothing *Louella Pizzuti*

Technical Buckstopper *Dave Johnson*

Managing Editor *Carol Westberg*

Production Editor *Toni Haskell*

Developmental Editors *Lorraine Anderson, Ann Cullen, Geta Carlson*

Editorial Assistant *Lenore Zelony*

Indexer *Ira Kleinberg*

Manager, Developer Press *David Krathwohl*

A R T & P R O D U C T I O N

Design/Art Direction *Joss Parsey*

Technical Illustration *J. Goldstein*

Production *Bruce Potterton*

Printing *Craftsman Press*

Film Preparation *FilmCraft*

Photographer *Ralph Portillo*

Circulation Management *Dee Kiamy*

Online Production *Cassi Carpenter*

R E V I E W B O A R D

Pete "Luke" Alexander

Tim "Sweetpea" Enwall

Larry "Cat Couch" Rosenstein

Andy "The Shebanator" Shebanow

S P E C I A L T H A N K S

Matt Deatherage

Cleo Huggins and Hal Rucker created the cover with help from timekeepers past and present. Cover photograph by Ralph Portillo.

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

CONTENTS

Dear Readers,

I just got back from sabbatical and it was great: I left and could hardly remember that I had ever worked for Apple, and now that I'm back I can hardly remember that I ever left. It's good, I think, to be fully where you are as you linger.

Upon my return, I wasn't quite sure how, in my advanced state of equilibrium, I was going to find a way for drivers and clock parts to live together in a logical editorial. So I decided to throw logic out the window and to stick with what I know: taking an analogy and stretching it. Here goes.

Clock parts are carefully crafted according to well-defined rules. Along with following the rules, creativity and craftsmanship are brought to bear, so some clocks are more pleasing, better functioning, and longer lasting than others. This is how it is with drivers, too. Respect for the rules, creativity, and craftsmanship combine to make a driver tick.

Clock parts as a group (or a watch) keep track of the moment-by-moment passage of time, freeing us to focus our attention on things more riveting. Similarly, a system-level driver lets your application focus on things more interesting (and useful) than hardware-specific details.

So much for the analogy. In this issue, Matt provides thorough coverage of the printer driver: what it does, how it does it, and how to write one for the Apple IIGS Zz tells even more about what your application can do with PostScript code to avoid perils posed by the LaserWriter driver. And if you decide to write your own driver, you can follow the legions before you and launch into assembly language, or you can follow Tim's lead and try C++.

On another topic, Scott A. Williams writes:

"On page 126, I believe that the **`AllocHeap`** method call to **`InitZone`** should have calls to **`GetZone`** and **`SetZone`** around it, like this:

```
THz  savedZone = GetZone ();
InitZone(nil, kNumDfltMasters, limitPtr, zonePtr);
SetZone(savedZone);
```

COMMENTS

We welcome timely letters to the editor, especially from readers wishing to react to articles that we publish in develop. Letters should be addressed to Louella Pizzuti, 20525 Mariani Ave., M/S 75-3B, Cupertino, CA 95014 (AppleLink 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).

"*Inside Macintosh*, volume II, page 29, says, '`InitZone` creates a new heap zone, initializes its header and trailer, and makes it the current zone.' It's the 'makes it the current zone' part that's the problem. Without the calls to `GetZone` and `SetZone`, any handles or pointers created after a call to the `AllocHeap` method would be allocated in the new heap created expressly for holding `PtrObjects` and not in the application heap where they belong."

Well, when Scott wrote he was right, and now he's sporting a fine new develop shirt. When you write, you will too.

[illegible]

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. •

WRITING A DEVICE DRIVER IN C++ (WHAT? IN C++?)

Most developers write device drivers in assembly language, rarely considering a higher level, object-based language such as C++ for such a job. This article describes some of the advantages of higher level languages over assembly and warns of some of the gotchas you may encounter if you write a driver in C++. An example of a device driver written in C++ follows a brief discussion of drivers in general.

TIM ENWALL

When you think of writing a device driver, your first reaction may be, "But I haven't brushed up on assembly language in some time." After taking a deep breath, you think of another approach: "Why can't I use a high-level language?" You can. One such language is C++.

In comparison with standard C, C++ offers some definite advantages, including ease of maintenance, portability, and reusability. You can encapsulate data and functions into classes, giving future coders an easier job of maintaining and enhancing what you've done. And you can take advantage of most (but not all) of the powerful features of C++ when you write stand-alone code.

You will run into a few gotchas, including the fact that polymorphism is available only if you do some extra work (for a definition of polymorphism, see develop, Issue 2, page 180). Because the virtual tables (vTables) reside in the jump-table segment, a stand-alone code resource can't get at the vTables directly (more on this topic later). You also have to deal with factors such as how parameters are passed to methods, how methods are called, how you return to the Device Manager, how you compile and link the DRVR resource, and how the DRVR resource is installed when the machine starts up. We'll tackle some of these obstacles as we work through the sample device driver presented later in this article.

TIM ENWALL, DTS engineer, is a four-year Apple veteran. He's done stints in Technical Resources and data base applications, and has answered the A/UX® hotline. Now his primary job purportedly revolves around IBM connectivity (although no one ever seems to ask him about it). The rest of his time is spent with networking and lower-level device managers. A Rocky Mountain native, he came to Cal Berkeley for an EE/CS degree. He likes bike riding, cooking, playing softball, and long talks with friends. Two Burmese cats—Bella and GBU (for the Good, the Bad, and the Ugly, pronounced "Boo")—guard him during his off hours. Two favorite books have helped

WHY C++?

When someone suggests writing a device driver in anything other than assembly language, the common reaction is, “But you’re talking to a *device!* Why would you want to use C++?”

For communication with devices, assembly language admittedly gets the job done in minimal time, with maximum efficiency. But if you’re writing something where code maintenance, portability, and high-level language functionality are just as important as speed and efficiency, a higher level language is preferable.

Not all device drivers actually communicate with physical devices. Many device drivers have more esoteric functions, such as interapplication communication, as in the sample driver in this article. (In fact, DAs are of resource type DRVR and behave exactly the same way device drivers behave. DAs are even created the same way.) For these kinds of device drivers, C++ is a great language to use because you can take advantage of all the features of a high-level language, plus most of the object-based features of C++. Finally, device drivers have some nice features that make them appealing for general usage:

- They can remain in the system heap, providing a common interface for any application to easily call and use.
- They get periodic time (if other applications are not hogging the CPU).

Good examples of nondevice drivers are the .MPP (AppleTalk®) driver and the .IPC (A/ROSE™ interprocess communication) driver. Both these drivers provide pretty high-level functionality, but neither directly manipulates a device as such (except for the very low-level AppleTalk manipulations of communication ports). Of course, if you were writing code to communicate quickly and efficiently to a modem, for example, assembly language might be the better choice, depending on your need for efficiency and timing. For the purposes of this article, any reference to a device driver includes both types of drivers.

Clearly, higher level languages have a place, but what about object-based languages? Object-based languages provide a great framework for encapsulation of data and functions and hence increase the ease of maintenance and portability (if used elegantly). One question still remains: Why C++?

Notables such as Bjarne Stroustrup and Stanley Lippman have pointed out some of the advantages C++ offers over conventional high-level languages. C++ offers great extensions, such as operator and function overloading, to standard C. C++ is much more strongly type checked than C, so it saves us programmers from ourselves. C++ classes offer a way to encapsulate data—and functions that operate on the data—within one unit. You can make different elements and functions “private” to objects of only one class or “public” to objects of every type. The private and public nature of data and member functions allows you to accomplish real encapsulation.

shape his outlook: *Light in August* by William Faulkner and *Native Son* by Richard Wright. But don’t let his serious side fool you. Watch out when he’s sitting across a poker table from you: he was brought up on cards, and he’s out to get that BMW 3.0CSi with a sunroof. Will you be the one to provide the down payment? •

COMPARING C++ AND ASSEMBLY LANGUAGE

	C++	Assembly Language
Pros	Portable	Fast
	Reusable	Efficient
	Easy to maintain	Compact
	Object-based design	Direct access to CPU
	High-level language features	
	Data encapsulation	
Cons	Three separate source files, multiple compiles	Not portable
	Speed inefficient	Hard to maintain
	Polymorphism difficult in stand-alone code	Lacking high-level language features such as loops and IF-THEN-ELSE

SOME LIMITATIONS

As noted, one valuable feature of C++, polymorphism, is not readily available when you write a device driver in C++. Other limitations involve working with assembly language, possible speed sacrifices, work-arounds for intersegment calls, and mangled procedure names.

POLYMORPHISM

Because a device driver is a stand-alone code resource, there is no "global" space or jump table. C++'s virtual function tables (vTables), which are the means to the polymorphism end, live in an application's global space. The loss of virtual tables is a limitation of stand-alone code, not a limitation of C++. Patrick Beard's article, "Polymorphic Code Resources in C++" (this issue), shows one way to work around this limitation. The work-around takes some extra work and is dependent on the current implementation of CFront, which may make future compatibility a problem. In the interests of clarity and compatibility, I have chosen not to use polymorphism for the example in this article.

ASSEMBLY-LANGUAGE MUCK

Another difficulty is that we have to get our hands assembly-language dirty. The Device Manager is going to call the device driver with a few registers pointing to certain structures, and we'll have to put those on the stack so the C++ routines can get to them. Specifically, `A0` points to the parameter block that is being passed, and

`A1` has a handle to the Device Control Entry for the driver. Having to do some assembler work is a limitation of the operating system; the toolbox doesn't push the parameters onto the stack (now if there were glue to do that—).

These registers must somehow make their way onto the stack as parameters to our routines because procedures take their parameters off the stack. When we've finished, we also have to deal with jumping to `jIODone` or plain `RTSing`, depending on the circumstances. For the simple driver shown in the example, we will in reality almost always jump via `jIODone` when finished with our routines. But, for drivers that wish to allow more than one operation at a time, the `Prime`, `Control`, and `Status` calls must return via an `RTS` to signal the Device Manager that the request has not been completed. The driver's routines should jump to `jIODone` only when the request is complete.

We must also decide whether or not to call a C++ method directly from the assembly language "glue." If we call the method directly, we have to put the "this" pointer on the stack because it's passed implicitly to all object methods. We also have to use the "mangled" name generated by the compiler and used by the linker. (If you haven't had the opportunity to see mangled names, you'll find they're a joy to figure out without the help of our friend Mr. Unmangle.) So, if we choose to call extern C functions, as the example does, we run into yet another level of "indirection" before we get to the real meat of the matter.

SPEED

Some might say we sacrifice speed as well as efficiency—and they're correct. In general, compilers can't generate optimally speed-efficient code. They can come close, but nothing even approaches how the human mind tackles some tricky machine-level issues. Thus, we're at the mercy of the compiler—the loss of speed is the result of the compiler's inefficiency.

You'll probably find the sample driver presented in this article pretty inefficient. But the trade-off is acceptable because speed isn't important in this case, and you can use all the features of an object-based language. In fact, in most instances you can limit assembly language to a few routines, which must be tightly coded, and use C++ for the rest.

MANGLED IDENTIFIERS

If you're familiar with C++, you've undoubtedly seen the visions of unreadability created by CFront. But, if you're still unfamiliar with C++ in practice, here's an explanation. CFront is simply a preprocessor that creates C code, which is passed to the C compiler. So CFront has to somehow take a function of the form

```
TDriver::iacOpen(ParmBlkPtr aParmBlkPtr)
```

and create a C function name the C compiler can understand. The problem is that when the linker complains, it will use the *mangled* name, which is hard to decipher.

Here's how it looks:

from MPW Shell document

```
unmangle iacOpen__7TDriverFP13ParamBlockRec
Unmangled symbol: TDriver::iacOpen(ParamBlockRec*)
```

It's clear why these names are referred to as mangled and unmangled. Fortunately, the unmangle tool provided with MPW allows you to derive the unmangled name from the mangled.

A SAMPLE C++ DRIVER

The sample driver that follows illustrates some of the issues involved in writing a device driver in general, and specifically in C++. The code is in the folder labeled C++ Driver on the *Developer Essentials* disc.

INTERAPPLICATION COMMUNICATION

The sample driver performs one basic function—interapplication communication (IAC)—under System 6. Under System 7 the services of this sample driver aren't necessary because IAC is built into the system. But the concepts presented here are still sound, and the driver works as well under System 7 as it does under System 6. The driver is installed at Init time with code that walks through the unit table looking for a slot.

CLASS STRUCTURE

The classes are fairly straightforward, serving as an example of how to use C++ to encapsulate data with methods without getting into some gnarly class hierarchies that would only obfuscate the point (and that aren't yet possible with stand-alone code). Two classes suffice: `TDriver` and `TMessage`. `TDriver` handles all the driving; it responds to each control and status call defined and handles opening and closing the driver. It keeps two simple data structures—an array of application names that have registered and an array of `TMessage` pointers that need to be received. `TMessage` handles the messages—who they're from, who they're addressed to, and what the message is. I think you'll find the declarations easy reading.

from TDriver.h

```
class TDriver: public HandleObject {
public:
      // Constructor and destructor.
      TDriver();
      ~TDriver();
      /* Generic driver routines. These are the only public interfaces we
       * show to the world.                                              */
      OSErr iacOpen(ParmBlkPtr oParmBlock);
      OSErr iacPrime(ParmBlkPtr pParmBlock);
      OSErr iacControl(ParmBlkPtr cntlParmBlock);
      OSErr iacStatus(ParmBlkPtr sParmBlock);
      OSErr iacClose(ParmBlkPtr cParmBlock);

private:
      // Control Routines.
      /* RegisterApp takes the string in iacRecord.appName and finds a slot
       * in the array for the name (hence it "registers" the application).
       * SendMessage sends a message from one application to another (as
       * specified by the iacRecord fields).
       * ReceiveMessage puts the message string into the iacRecord.msgString
       * field if there's a message for the requesting application.
       * UnregisterApp removes the application's name from the array (hence
       * the application is "unregistered").                       */
      short       RegisterApp(IACRecord   *anIACPtr);
      short       SendMessage(IACRecord   *anIACPtr);
      short       ReceiveMessage(IACRecord*anIACPtr);
      short       UnregisterApp(IACRecord *anIACPtr);

      // Status Routines
      /* WhosThere returns the signature of other applications that
       * have registered.
       * AnyMessagesForMe returns the number of messages waiting for the
       * requesting application in iacRecord.actualCount.         */
      void        WhosThere(IACRecord     *anIACPtr);
      Boolean     AnyMessagesForMe(IACRecord    *anIACPtr);

      // Message array handling routines.
      /* GetMessage gets the TMessPtr in fMessageArray[signature].
       * SetMessage sets the pointer in fMessageArray[signature] to
       * aMsgPtr.                                             */
      TMessPtr    GetMessage(short signature);
      void        SetMessage(short index, TMessPtr    aMsgPtr);

      // AppName array handling routines.
      /* GetAppName gets the application name in fAppNameArray[signature].
       * SetAppName sets the application in fAppNameArray[signature]
       * to anAppName.                                              */
      char        *GetAppName(short signature);
      void        SetAppName(short signature, char *anAppName);
      /* We keep an array of applications that can register with the
       * driver. I've arbitrarily set this at 16. We also keep an array
       * of TMessage pointers to be passed around. This is also arbitrarily
       * set at 16. In the future, I'd probably implement this as a list of
       * messages.                                           */
      char        fAppNameArray[kMaxApps] [255];
      TMessPtr    fMessageArray[kMaxMessages];
};
```

from TMessage.h

```
class TMessage {
public:
            /* Constructor and destructor. Constructor will build the message
             * with the appropriate data members passed in. */
            TMessage(char *message, short senderSig, short receiverSig);
            ~TMessage();

            /* Two Boolean functions that simply query the message to see
             * if the message is destined for the signature of the
             * Requestor. Nice example of function overloading in the
             * one case I just wanted to return true or false; in the other
             * case I wanted to return who the message was from and the actual
             * message string. This is also nice because we have only one
             * public member function returning any private information. */
      Boolean     IsMessageForMe(short sigOfRequestor);
      Boolean     IsMessageForMe(short sigOfRequestor, short *senderSig,
                                 char *messageString);

private:
      /* GetSenderSig returns fSenderSig.
       * SetSenderSig sets fSenderSig to signature.                */
      short       GetSenderSig();
      void        SetSenderSig(short signature);

      /* GetReceiverSig returns fReceiverSig.
       * SetReceiverSig sets fReceiverSig to signature.            */
      short       GetReceiverSig();
      void        SetReceiverSig(short signature);

      /* GetMessageString returns fMessageString.
       * SetMessageString sets fMessageString to msgString.        */
      char        *GetMessageString();
      void        SetMessageString(char *msgString);
      // Private data members. Again, we keep storage for the string here.
      short       fSenderSig;
      short       fReceiverSig;
      char        fMessageString[255];
};
```

The only remaining structure worthy of note is the `IACRecord` structure. This structure is passed in the `csParam` field of the parameter block pointers passed to the driver. Essentially the `IACRecord` structure contains all the control information, or returns all the status information, the application needs to communicate—the signatures of the sender and receiver, the message and application name strings, and a couple of other control fields.

from IACHeaders.h

```
struct IACRecord  {
      // Signature number of application sending/receiving.
      short mySignature;

      // Signature of app that's either sent a message or
      // of app to which the current app is sending.
      short partnerSig;

      // Index to cycle through the apps that have registered.
      short indexForWhosThere;

      // Nonzero if messages there for recipient.
      short actualCount;

      // Message string being sent or received.
      char* messageString;

      // String to register as.
      char  *appName;
};
```

REGISTERING WITH THE DRIVER

To use the driver, an application registers itself with the driver, thus signifying that the application is able to receive and send messages. The driver returns a unique signature for the application to use throughout the communication session. A second (or third, or fourth) application also registers and communicates with other applications by sending and receiving messages using the correct signature. When an application is finished, it simply unregisters itself. Here are four of the methods that do most of the work:

from TDriver.cp

```
/********************************Comment****************************************
* TDriver::RegisterApp looks to see if there's an open "slot".  If so, it sets
* the new AppName for that "slot" and returns the "slot" as the signature.  If it
* couldn't find any open "slots" then it returns the kNoMore error.
 ********************************End Comment***********************************/
short
TDriver::RegisterApp(IACRecord *anIACPtr)
{
short       i = 0;
short       canDo = kNoMore;

while ((i < kMaxApps) && (canDo == kNoMore))
      {
      if((this->GetAppName(i))[0] == kZeroChar)
            {
            canDo = kNoErr;
            anIACPtr->mySignature = i;
            this->SetAppName(i,anIACPtr->appName);
            }
      i++;
      }
return (canDo);
} //  TDriver::RegisterApp

/********************************Comment****************************************
* TDriver::SendMessage has to instantiate a new message object.  It also has to
* remember that message for later when someone tries to receive it.  To remember
* it, the TDriver object places it in the message pointer array.  If it couldn't
* find an open "slot" in the array, it returns the error kMsgMemErr, meaning it
* has no memory to store the pointer to the message and hence the message didn't
* get sent. Since the TDriver object is creating a new TMessage, it will destroy
* the TMessage when the time comes.
 ********************************End Comment***********************************/
short
TDriver::SendMessage(IACRecord *anIACPtr)
{
TMessPtr    aMsgPtr;
short       canDo = kNoMore;
short       i = 0;

aMsgPtr = new TMessage(anIACPtr->messageString, anIACPtr->mySignature,
            anIACPtr->partnerSig);
```

```
if(aMsgPtr)
      {
      while ((i < kMaxMessages) && (canDo == kNoMore))
            {
            if(this->GetMessage(i) == nil)
                  {
                  this->SetMessage(i, aMsgPtr);
                  canDo = kNoErr;
                  }
            i++;
            }
      if (canDo == kNoMore)
            delete aMsgPtr;
      } // if aMsgPtr
else
      canDo = kMsgMemErr;
return (canDo);
} //  TDriver::SendMessage

/**********************************Comment**********************************
* TDriver::ReceiveMessage finds any messages for the application whose
* signature is mySignature. It first checks to see if there are any
* messages. If so, it gets the message and asks the TMessage object to
* return the message string. Then it copies the message string to the
* calling application's message buffer, puts the sender's signature in
* "partnerSig", and puts the sender's application name in appName.
* It then sets the "slot" in the message array to nil and disposes of the
* TMessage object. If there were messages, it returns the kYesMessagesForMe
* value; otherwise it returns kNoMore.
 **********************************End Comment*****************************/
short
TDriver::ReceiveMessage(IACRecord *anIACPtr)
{
TMessPtr          aMsgPtr;
short             sender;
char              *bufP = nil;

if(this->AnyMessagesForMe(anIACPtr))
      {
      aMsgPtr = this->GetMessage(anIACPtr->actualCount);
      (void) aMsgPtr->IsMessageForMe(anIACPtr->mySignature,&sender,bufP);
      anIACPtr->partnerSig = sender;
      tseStrCpy(anIACPtr->messageString,bufP);
      tseStrCpy(anIACPtr->appName, this->GetAppName(anIACPtr->partnerSig));
      this->SetMessage(anIACPtr->actualCount,nil);
      delete aMsgPtr;
      return (kYesMessagesForMe);
      }
else
      return(kNoMore);
} // TDriver::ReceiveMessage

/************************Comment***************************
* TDriver::UnregisterApp receives all the messages for the
* application that is unregistering. Those messages will
* just get thrown away. So, all the messages destined for
* it are disposed of, and then it sets the name to '\0' so
* others can play.
 ***********************End Comment***********************/
short
TDriver::UnregisterApp(IACRecord *anIACPtr)
{
char          zeroChar = kZeroChar;

// Gotta delete those suckers.
while (this->ReceiveMessage(anIACPtr) == kYesMessagesForMe)
      ;
// Zero the name so others can play.
this->SetAppName(anIACPtr->mySignature,&zeroChar);
return (kNoErr);
} // TDriver::UnregisterApp
```

ASSEMBLY WRAPPED AROUND EXTERN "C", WRAPPED AROUND C++
When you open the C++ Driver folder *(Developer Essentials* disc), you see many source files, including the files DriverGlue.a and DriverWrapper.cp.
The assembly glue performs three main functions:

- Pushing the appropriate registers onto the stack.
- Returning to the Device Manager in the proper manner.
- Setting up the DRVR resource with the appropriate routine offsets in the offset fields.

The first two functions were covered earlier, but the third deserves some further note.

If you just glance at the MPW® manual, creating the DRVR resource seems like a breeze. There's an entire section on it, right? Wrong. The section on building DRVRs is a good excursion into how to compile and link a DA (how they got to be DRVRs we'll never know), but only serves to mislead when it comes to "real" DRVR resources.

MPW provides a great run-time library for DAs called DRVRRuntime.o, and it also provides a resource template that `rez` can use to create the final DRVR resource. The DRVW resource template included in MPWTypes.r even provides a nice programming description of the DRVR resource, but falls short when you delve into specifying the routine offsets every "device driver" needs to its `Open/Prime/Control/Status/Close` routines. The DRVRRuntime.o library simply provides jump statements to the appropriate pc-relative address for the `DRVROpen`, `DRVRPrime`, `DRVRControl`, `DRVRStatus`, and `DRVRClose` routines. Hence, the offsets are only 4 bytes apart, and right there the DRVR is hosed because the Device Manager has no way to jump to, say, the device driver's control routine.

For example, say an `Open` routine is 48 bytes long. If you use the DRVW template, the DCE header will contain 0 as the offset for the `Open` routine, 4 as the offset for the `Prime` routine, 8 as the offset for the `Control` routine, and so on. When the Device Manager goes to call the `Control` routine, it will jump 8 bytes into the `Open` routine and start executing there—*not* what you had intended. The only recourse is to use DriverGlue.a as an entry point and define the offsets at the beginning of the assembly file (calculating the offsets appropriately). So much for having `rez` help out; maybe the assembler will be more helpful.

The "main" procedure, created to compensate for `rez`'s ineffectiveness, looks like this:

from DriverGlue.a
```
HEADERDEF   PROC        EXPORT

            IMPORT      TSEPrime
            IMPORT      TSEOpen
            IMPORT      TSEControl
            IMPORT      TSEStatus
            IMPORT      TSEClose
TSEStartHdr DC.W        $5F00       ; Turn the proper bits on
                                    ; dNeedLock<6>, dNeedGoodbye<4)
                                    ; dReadEnable<3>, dWritEnable<2>
                                    ; dCtlEnable<1>, dStatEnable<0>
            DC.W        $12C        ; 5 seconds of delay (if dNeedTime = True)
            DC.W        0           ; DRVREMask (for DAs only)
            DC.W        0           ; DRVRMenu (for DAs only)
            DC.W        TSEOpen-TSEStartHdr     ; Offset to open routine.
            DC.W        TSEPrime-TSEStartHdr    ; Offset to prime routine.
            DC.W        TSEControl-TSEStartHdr  ; Offset to control routine.
            DC.W        TSEStatus-TSEStartHdr   ; Offset to Status routine.
            DC.W        TSEClose-TSEStartHdr    ; Offset to Close routine.
DC.B          '.TimDriver'; Driver name.
ALIGN         4           ; Align to next long word.
ENDP
```

It would be ideal to jump straight from the assembly language glue to the C++ `TDriver` methods and let the object do the work. Unfortunately, it's not that easy. First, we would have to allocate the `TDriver` object's space and put it into the `dCtlHandle` slot in the DCE. Second, we would have to do additional work in the glue code because each method implicitly expects that a pointer to the "this" object (the this pointer) will be passed on the stack. We would also have to stuff the `dCtlStorage` field into the "this" pointer address register.

The assembler isn't smart enough to figure out a directive like `JMP TDRIVER::IACOpen`. We could use the mangled name of the method and import that name at the start of the glue code, but all that seems a little too much for our assembly-naive minds. Apparently, then, the assembler isn't of much help either.

Instead, we'll resort to calling regular global C++ functions. We'll declare the functions as extern "C" functions so the compiler won't mangle the names, but will still compile them as regular C++ functions (because C++ is backward compatible with regular C). We end up with the following:

from DriverGlue.a

```
***************************** TSEOpen ***************************************
* This routine (and all like it below) performs three basic functions:
* 1. Pushing the parameter block (A0) and the pointer to the DCE (A1)
* on the stack.
* 2. Testing to see whether the immediate bit was set in the trap word and,
* if so, RTSing.
* 3. Testing the result in D0. If it's 1, the operation hasn't completed
* yet so we just want to RTS. If it's NOT 1, then we'll jump through
* jIODone.
* I put the standard procedure header in just so you'd see another example of
* it in use. I found Sample.a to be most helpful in much of what I did here.
*****************************************************************************
TSEOpen     PROC  EXPORT            ; Any source file can use this routine.

StackFrame  RECORD{A6Link},DECR     ; Build a stack frame record.
Result1     DS.W        1                 ; Function's result returned to caller.
ParamBegin  EQU         *                 ; Start parameters after this point.
ParamSize   EQU         ParamBegin-*; Size of all the passed parameters.
RetAddr     DS.L        1                 ; Placeholder for return address.
A6LinkDS.L        1                 ; Placeholder for A6 link.
```

```
LocalSize    EQU        *                  ; Size of all the local variables.
             ENDR                          ; End of record definition.

             WITH       StackFrame         ; Cover our local stack frame.
             LINK       A6,#LocalSize      ; Allocate our local stack frame.

             MOVEM.L    D1-D3/A0-A4,-(A7) ; Save registers (V1.1A).
             MOVE.LA1,-(A7)          ; Put address of DCE onto stack.
             MOVE.LA0,-(A7)          ; Put address of ParamBlock onto stack.
             JSR        TSDRVROpen         ; Call our routine.
             ADDQ.W#$8,A7      ; Take off A0 and A1 we pushed.
ADDA.L       #ParamSize,SP     ; Strip all the caller's parameters.
             MOVEM.L    (A7)+,D1-D3/A0-A4 ; Restore registers (V1.1A).
             SWAP       D0                 ; Save result in MostSig Word.
             MOVE.WioTrap(A0),D0     ; Move ioTrap into register to test.
             SWAP       D0                 ; Back again.
             BTST       #(noQueueBit+16), D0    ;Test the bit.
             BNE.S      OpenRTS            ; If Z = 0, then noQueueBit.
                                           ; Set   branch.
             CMP.W      #$1,D0       ; Compare result with 1.
             BEQ.S      OpenRTS            ; Not equal to zero so RTS.
             UNLK       A6                 ; Destroy the link.
             MOVE.LjIODone,-(A7)     ; Put jIODone on the stack.
             RTS                           ; Return to the caller.

OpenRTS      UNLK       A6                 ; Destroy the link.
             RTS                           ; Return to the caller.
             DbgInfo    TSEOpen            ; This name will appear in the debugger.
             ENDP                          ; End of procedure.
```

These global functions will do only some minor work that amounts to getting a pointer to the driver and calling the appropriate method. The `Open` routine does have to instantiate the object and install it into the `dCtlHandle` field of the DCE for subsequent retrieval. And the `Close` routine has to reverse these effects and dispose of the memory allocated by the `Open` procedure. All in all, however, the code is straightforward and, again, easy to follow.

from DriverWrapper.cp

```
/********************************Comment*************************************
* TSDRVROpen is called by the assembly TSEOpen routine. It in turn will simply
* turn around and call the TDriver::IACOpen method after some setup. This
* routine must instantiate the TDriver object. We'll be good heap users and move
* the object (handle) hi. If we get an error, we'll return MemErr, mostly for
* debugging purposes. Declared as extern "C" in DriverWrapper.h
 ******************************End Comment**********************************/

OSErr
TSDRVROpen(ParmBlkPtr oParmBlock,DCtlPtr tsDCEPtr)
{
TDrvrPtr    aDrvrPtr;
OSErr       err;

// Create TDriver object.
aDrvrPtr = new(TDriver);

// Make dCtlStorage point to it.
tsDCEPtr->dCtlStorage = (Handle) aDrvrPtr;
if(tsDCEPtr->dCtlStorage)
      {
      MoveHHi(tsDCEPtr->dCtlStorage);
      HLock(tsDCEPtr->dCtlStorage);
      aDrvrPtr = (TDrvrPtr) tsDCEPtr->dCtlStorage;
      err = aDrvrPtr->iacOpen(oParmBlock);// Call the iacOpen() method.
      HUnlock(tsDCEPtr->dCtlStorage);
      return(err);
      }
else
      return MemError();
}

/********************************Comment**********************************
* TSDRVRControl is called by the assembly TSEControl routine. It in turn
* simply turns around and calls the TDriver::IACControl method after locking
* the object. This essentially just locks the handle whose master pointer
* points to the object and then calls the appropriate method. When done,
* TSDRVRControl unlocks the handle.
*******************************End Comment*******************************/
OSErr
TSDRVRControl(ParmBlkPtr cntlParmBlock,DCtlPtr tsDCEPtr)
{
TDrvrPtr    aDrvrPtr;
OSErr       err;
HLock(tsDCEPtr->dCtlStorage);                   // Lock the storage handle.
aDrvrPtr = (TDrvrPtr) tsDCEPtr->dCtlStorage;    // Object pointer = master
                                                // pointer.
err = aDrvrPtr->iacControl(cntlParmBlock);// Call the iacControl() method.
HUnlock(tsDCEPtr->dCtlStorage);                 // Unlock the handle.
return(err);
}
```

We now have three "kinds" of source files: (1) the assembly language glue, (2) the global C++ functions declared as extern "C" so the names will be normal (our driver "wrapper" functions), and (3) the C++ object methods. Having an assembly routine call a global C++ function, which calls a C++ method, seems like quite a hassle, but avoiding having to do the whole thing in assembly language is well worth the effort, especially with our friend Mr. Linker to put everything together.

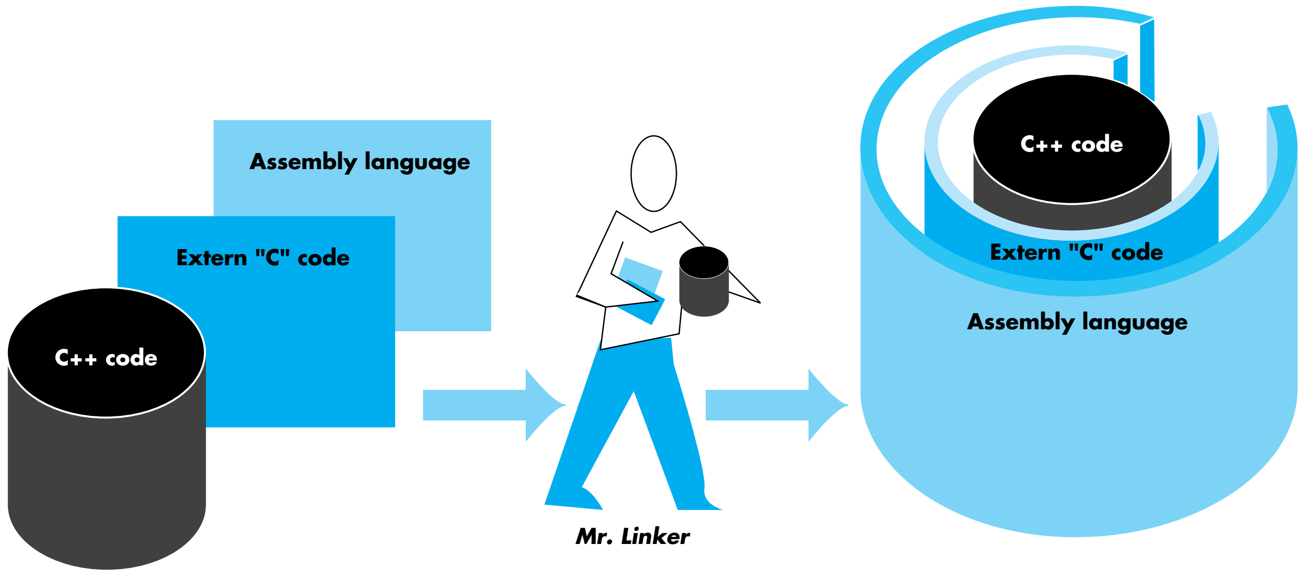


Figure 1
Using the Linker to Create the DRVR Resource

CREATING THE DRVR RESOURCE ITSELF

The linker handles the entire task of creating the DRVR resource in the driver resource file. Here, again, there are some caveats about usage. First, you need to make sure that the first elements in a DRVR are the flags and offsets, so the first procedure in the assembly language file just defines these with `DC.W` instructions. Second, you have to tell the linker where the first procedure is, so you specify the name with the -m option (in this case -m HeaderDef). Third, you have to give the DRVR resource the name you want the resource to have, so you use the -sn option to define this. Finally, you want to specify the resource attributes at link time, so you specify the DRVR resource as being locked, in the system heap, and preloaded. The link line looks like this:

from iacDriver.make
```
Link -rt DRVR=75 -m HEADERDEF -sn "Main=.TimDriver" ¶
      -c 'TSEN ' -t 'DRVR' ¶
      -ra ".TimDriver"=resSysHeap,resLocked,resPreLoad ¶
      {CPOBJECTS} ¶
      "{Libraries}Interface.o" ¶
      -o iacDriver.DRVR
```

Sometimes the compiler (or CFront) does things behind your back that are completely frustrating, even if you're a careful programmer. The first time I tried to link the driver together, the linker complained that data initialization code had not been called. I knew there was no "data initialization" code being called because I had compiled a stand-alone code resource. I scratched my head because I knew I didn't have any globals anywhere in my code. Then I remembered, "Oh yeah, the compiler puts string constants in the global segment." The MPW manual explains the -b option, and eventually that option worked to solve the problem. I say "eventually" because I ran into another case where the compiler helped me out without my knowing.

Definitions for new and delete are included in the CPlusLib.o library. In this case, CFront calls these functions for every constructor. Even if you define your own new and delete functions, the linker still will include the CPlusLib.o versions of the functions in the global segment. The linker then *still* thinks it has global data that hasn't been initialized.

The solution to the problem is to define your own external "C" functions (an indicator to the compiler to use regular C calling conventions, but still part of your C++ code) with the mangled names for new and delete. You'll have to declare the functions as returning a void pointer or handle. The declarations look like this:

from iacGlobalNewDel.cp
```
/* unmangle __nw__FUi
 * Unmangled symbol: operator new(unsigned int)
 * We return a void * because new returns a pointer.  */
void *__nw__FUi(unsigned int size)
```

As an alternative to defining your own external "C" functions, you could use the A5 global library routines described in Technical Note #256, Stand-Alone Code, *ad nauseam*. You could then use globals as well as the default code for new and delete.•

```
/* unmangle __dl__FPv
 * Unmangled symbol: operator delete(void *)
 * We return void just for clarity.                                */
void __dl__FPv(void *obj)

/* unmangle __nw__12HandleObjectSFUi
 * Unmangled symbol: static HandleObject::operator new(unsigned int)
 * We return a void ** because this version of new should return
 * a handle.                                                  */
void **__nw__12HandleObjectSFUi(unsigned int size)

/* unmangle __dl__12HandleObjectSFPPv
 * Unmangled symbol: static HandleObject::operator delete(void **) */
void __dl__12HandleObjectSFPPv(void **aHandle)
```

You have to use the mangled names because that's how they were compiled into the CPlusLib.o library. Fortunately, you can now eliminate the CPlusLib.o library from your list of libraries. Once past these two global obstacles—string constants placed in the global segment and `new/delete` operators called from constructors—the linker passes the sample code through with flying colors.

BUILDING THE 'INIT' TO INSTALL THE DRIVER

Now that the DRVR resource and code are finished, how do you use it? The first order of business is to install the driver into the UnitTable. The listing for the code that does the installation appears on the next page. This code opens the resource file where the DRVR resides, looks for an open "slot" in the UnitTable starting from the rear of the UnitTable, opens the resource, changes the resource ID to match the UnitTable slot, calls `OpenDriver`, detaches the resource, and changes the DRVR resource ID back to what it was before beginning. The few steps that need explanation are finding the slot in the UnitTable, calling `DetachResource`, and calling `OpenDriver`.

Why do you have to find an open slot in the UnitTable? You want to make sure the driver gets installed. If there's a resource ID conflict (and hence a slot conflict in the UnitTable), you can't be sure whether the driver will clobber the existing one or won't get installed at all. Thus, you could rely on just calling `OpenDriver` with the DRVR resource ID, but that wouldn't be very cooperative of you. So you look for an open slot, which boils down to looking for a nil address in the UnitTable, starting at the back of the UnitTable where open slots are most likely to exist (the system uses up slots at the beginning of the UnitTable). If the contents of the address are nil, you can install the driver into that slot.

from installDriver.c
```
short
lookForSlotInUnitTable()
{
short          slot;
Ptr            theBass;
long           *theVoidPtr;
Boolean foundSlot = false;

/* Set up variables based on contents of low-memory global
 * locations.  DTS tells people not to rely on low-memory
 * globals, but we really need these two low-memory
 * globals to do our work. So, there is a compatibility
 * risk we have to be aware of.                         */

slot = *((short *)(UnitNtryCnt)) - 1;
theBass = (Ptr) (*((long *) (UTableBase)));

// We step back to 48 because 0-47 are taken.
while(slot>48 && !foundSlot)
      {
      theVoidPtr = (long *)(theBass + (4L * slot));

      if(*theVoidPtr == nil)
        foundSlot = true;

      slot -= 1;
      }

slot += 1;
if(!foundSlot)
  slot = 0;
return slot;
}
```

Why do you call `DetachResource`? *Inside Macintosh*, volume V, page 121, says, "`DetachResource` is also useful in the unusual case that you don't want a resource to be released when a resource file is closed." The example is such a case. When the Init is loaded and executed by the Init 31 mechanism, the resource file in which the Init resides is opened. When the Init has been executed, the resource file is closed, and the Resource Manager goes around and cleans up any of the resources in the resource map that are known to be allocated. `DetachResource` replaces the handle in the resource map with nil, so the Resource Manager thinks it doesn't have to clean up that handle.

Thanks to Pete Helme for the `installDriver.c` code. •

Why do you call `OpenDriver` instead of `_DrvrInstall`? Essentially that's because `OpenDriver` does the correct thing and `_DrvrInstall` doesn't. When you call `_DrvrInstall` with a handle to the driver, `_DrvrInstall` does most of the work, but it forgets to put the handle to the driver into the `dCtlDriver` field of the DCE and effectively makes the driver unreachable. `_OpenDriver` has no such problem, and it works correctly. Alternatively, you could use `_DrvrInstall` and then put the handle to the driver into the DCE. The installation code looks like this:

from installDriver.c

```
void
changeDRVRSlot(short slot)
{
Handle      theDRVR;
short             err, refNum;
char              *name, DRVRname[256];
short             DRVRid;
ResType           DRVRType;

name = "\p.TimDriver";

if(slot != 0) {
      theDRVR = GetNamedResource('DRVR', name);
      GetResInfo(theDRVR, &DRVRid, &DRVRType, &DRVRname);
      SetResInfo(theDRVR, slot, 0L);

      err = OpenDriver(name, &refNum);
      if(err == noErr)
            {
            /* Detach the resources from the resource map. */
            DetachResource(theDRVR);

            }
      /* Restores the previous resource attributes so they don't change
       * from start-up to start-up. We just want the in-memory copy to
       * have a different ID not our resource in the file. */
      theDRVR = GetNamedResource('DRVR', name);
      SetResInfo(theDRVR, DRVRid, nil);
      }
}
```

This code needs to be compiled with the `-b` option as well because it's a stand-alone code resource like the DRVR, and you have to have everything in one resource. For the example, we chose to make the installation code an Init so that the driver will install at system start-up time and so that any application can access it. You also *must* make sure that the resource has the resource attribute `resLocked`. The Init must be locked at start-up time in case anything in the Init code moves memory. If anything in the Init does move memory, you come back to some random place in the system heap because the Init resource has been moved. This is a particularly painful (and time-consuming) gotcha.

PUTTING IT ALL TOGETHER

The final goal is to have one file that contains all the necessary resources. At this point you have all the code resources you need: the Init and the DRVR. You may need one additional resource, depending on how large the driver is and how much of the system heap you need. If you need more than 16K, you have to create the `sysz` resource and put that in the file. Fortunately, the `sysz` resource is simple to define; it looks like this:

from iacDriver.r

```
include "iacDriver.DRVR"; /* Include the DRVR resource. */
include "installDriver";  /* Include the INIT resource. */

type 'sysz' {   /* This is the type definition. */
  longint;      /* Size requested (see IM V, page 352).*/
};

resource 'sysz' (0,"",0) { /* This is the declaration.  */
  0x00008000  /* 32 * 1024 bytes for sysz resource.     */
};
```

Now that you have all the components, you let `rez` do the work of moving the Init and DRVR resources into one file. Fortunately you can include resources from other resource files with the "include" directive (see chapter 11, page 309, in the MPW manual for a discussion of `rez`).

from iacDriver.make

```
rez iacDriver.r -c TSEN -t INIT -a -o iacDriver
```

CALLING THE DRIVER FROM AN APPLICATION

The example also includes several routines you might run from a client application to use the sample driver. *Developer Essentials* contains two sample applications that use these routines to register and send or receive messages. (Don't get your hopes up, though. This is just Sample.c modified, so the light will turn off and on via control from a second application.)

DESIGN DECISIONS

Now that you've learned about this "device driver" in particular, and more about drivers in general, we can discuss some of the trade-offs required.

WHAT TO DO WITH JIODONE, AND WHEN

Most of the time, the device driver should jump to `jIODone` so the Device Manager will handle the housekeeping tasks of marking the driver as "unbusy" and calling the completion routine. However, a few exceptions are noted throughout the chapter on the Device Manager (*Inside Macintosh*, volume II, chapter 6). You *don't* want to jump to `jIODone` (just `RTS`ing instead) in these situations:

- When an operation you started is not yet complete (that is, an operation that will interrupt you when it *is* complete).
- When you get a `KillIO` request.
- When you get called immediately (that is, bit #9, the `noQueueBit`, is set in the `ioTrap` word and calls usually look like `_Read, IMMED)`.

Speaking of the immediate bit, you'll find that most drivers don't guard against reentrancy. This is a problem when callers try to make `Immediate` calls to the driver. If you don't want people making `Immediate` calls to the driver, you have to specify in the documentation that callers may not call this device driver immediately; otherwise, the results will be indeterminate. On the other hand, if you do want to allow `Immediate` calls, one simple way to guard against most types of reentrancy problems is to set some flag within the driver and then either (1) return without performing the immediate action requested or (2) save the state of the other operation, perform the `Immediate` call, and return. In either event, remember to return via an `RTS` for all `Immediate` calls.

MULTIPLE OUTSTANDING REQUESTS

You may want a driver to be able to handle many `_Read` and `_Write` requests at the same time, and not one at a time. This driver can handle only one request at a time. If no messages are waiting *when requested*, for example, the caller is told there are no messages. In many cases, however, you want to keep that request around until there *is* a message. To handle this case, you have to do some more work. Essentially, you have to dequeue the request from the Device Manager's queue, queue it up in some internal list of your own, and then satisfy the requests when they are finished. You have to perform the functionality of `jIODone` yourself as well, because you'll be handling the operations yourself. You're also operating behind the Device Manager's back to some extent because you're dequeueing requests from the I/O queue yourself.

_READ AND _WRITE OR _CONTROL OPERATIONS

If you use `_Read` and `_Write`, you can't pass in `csParam`. The trade-off we made in the example was that `csParam` would point to a structure that gave us more control over, and a more elegant solution to, sending and receiving messages to and from the proper place. If you use `_Read/_Write`, you have to format the `ioBuffer` to contain all the information for the messages, and that means encoding the sender and receiver signatures in with the actual message. One disadvantage of this trade-off is that the method may fail in the future in the world of virtual memory. Virtual memory watches the `_Read` and `_Write` traps and makes sure the memory addressed by `ioBuffer` stays in physical memory, but it neglects to do the same for `csParam`. Hence, the `IACRecord` structure (and pointers within that structure) may or may not be in physical RAM at the time of the call. If this happens at interrupt time and a page fault occurs, you're completely hosed.

TMESSAGE OBJECTS

Finally, the sample driver isn't very space friendly. The `TMessage` objects are allocated by `NewPtrSys`, and hence will fill up the system heap with locked pointers. The good news is that `TMessage` objects probably don't live for very long. The bad news is that the heap may still become fragmented. So, another design decision you could make would be to derive from `HandleObject` and take into consideration dereferences of handles. You may want to try that as an exercise.

For More Information

Inside Macintosh, volume II, chapter 6, "The Device Manager."

Stanley Lippman: *The C++ Primer,* Addison-Wesley, 1989.

Bjarne Stroustrup: *The C++ Programming Language,* Addison-Wesley, 1987. •

SUMMING UP

To summarize: C++ *can* be used to write a device driver that operates under some basic restrictions. We successfully built a couple of stand-alone classes that can be modified and kept up separately. The classes present a clear definition of roles and hide data as cleanly as possible. We chose not to use polymorphism in the code, although we certainly could have done so—with a little extra work and the possibility of future incompatibilities (again, see "Polymorphic Code Resources in C++," by Patrick Beard, in this issue).

Because of limitations of the operating system and development system, we have to incorporate some assembly language, and some global C++ functions, into whatever we write. We discussed some of the design trade-offs you must inevitably make and went into some depth on several of the trickier aspects of writing a device driver—what to do with `jIODone`, how to use the assembler to best advantage, compiling and linking the stand-alone code so it does the right thing, creating an Init that installs the driver at system start-up time, and using `rez` to create the eventual resource file.

C++ allows you to encapsulate data with functions, thus making it easier to maintain code and port the code to other platforms. Some nifty language features, such as function overloading and strong type checking, come with C++. If you're writing a device driver that doesn't depend on speed and efficiency, C++ is a good choice of languages.

Thanks to Our Technical Reviewers
Brian Bechtel and Jack Palevich •

POLYMORPHIC CODE RESOURCES IN C++

The C++ programming language supports data abstraction and object programming. Until now, using C++ to its full capacity in stand-alone code has not been possible. This article demonstrates how you can take advantage of two important features of C++, inheritance and polymorphism, in stand-alone code. An example shows how to write a window definition function using polymorphism.

PATRICK C. BEARD

In object programming, polymorphism gives programmers a way to solve problems by beginning with the general and proceeding to the specific. This process is similar to top-down programming, in which the programmer writes the skeleton of a program to establish the overall structure and then fills in the details later. Polymorphism differs from top-down programming, however, in that it produces designs that are reusable outside the context of the original structure. The attractiveness of reusable code is one of the reasons object programming is catching on.

The shape hierarchy shown in Figure 1 is one of the most frequently cited examples of polymorphism.

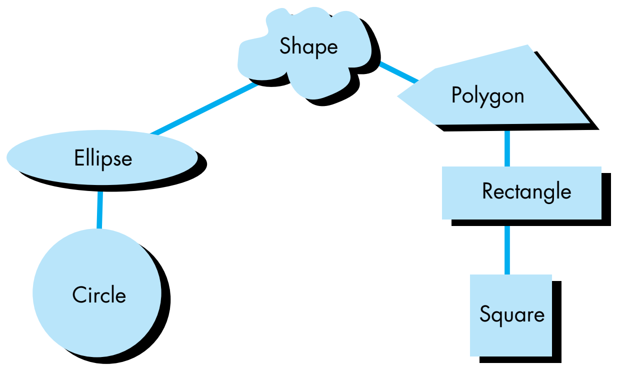


Figure 1
Shape Hierarchy

PATRICK BEARD of Berkeley Systems, Inc., is a totally rad dude, living in a world somewhere between hard-core physics and fantasy. He prepared for this lifestyle by getting a B.S.M.E. at the University of California, Berkeley. He claims native Californian status, although he's from Illinois. A programming addict who relishes treading the very edge of what's possible, Pat dreams of writing his own programming language so he can really express himself. Meanwhile, he has written screen-savers and sundry compiler hacks, and has helped develop a Macintosh talking interface for the blind. He's a jazz musician (looking for a rhythm section—any takers?), a snow skier, snowboarder, and

The most general concept is the shape; all objects in the hierarchy inherit attributes from the shape. Area, perimeter, centroid, and color are attributes common to all shapes. Notice that the hierarchy proceeds from the general to the specific:

- Polygons are shapes with a discrete number of sides.
- Rectangles are polygons that have four sides and all right angles; squares are rectangles having all equal sides.
- Ellipses are shapes that have a certain mathematical description; circles are ellipses whose widths equal their heights.

In C++, concepts are represented as classes. The more abstract the concept, the higher in the inheritance hierarchy the concept resides. Two key C++ features support polymorphism: inheritance and virtual member functions. We can use these to develop more concretely specified shapes and ask questions of any shape about its area, perimeter, or centroid.

The virtual functions provide a protocol for working with shapes. Here is an example of the shape hierarchy as it could be represented in C++:

```
class Shape {
public:
      virtual float area();         // Area of the shape.
      virtual float perimeter();    // Its perimeter.
      virtual Point centroid();     // Its centroid.
};

class Ellipse : public Shape {
public:
      virtual float area();         // Area of the shape.
      virtual float perimeter();    // Its perimeter.
      virtual Point centroid();     // Its centroid.
private:
      Point center;                 // Center of ellipse.
      float height;                 // How high.
      float width;                  // How wide.
};

class Circle : public Ellipse {
public:
      virtual float area();         // Area of the shape.
      virtual float perimeter();    // Its perimeter.
      virtual Point centroid();     // Its centroid.
};
```

skateboarder, whose motto in life is "Stop and breathe from time to time." He never puts anything away, fearing an inability to find stuff when he needs it; the piles are growing at an alarming rate. However, he swears his brain is organized and that he knows where everything is, except Tech Note #31. •

In this implementation, a circle is an ellipse with the additional constraint that its width and height must be equal.

Once an object of a type derived from `Shape` has been instantiated, it can be manipulated with general code that knows only about shapes. The benefit is that, having written and debugged this general code, you can add more kinds of shapes without having to alter the general code. This eliminates many potential errors.

IMPLEMENTATION IN MPW C++

MPW C++ is a language translator that translates C++ to C. Programs are compiled by first being translated to C, after which the MPW C compiler takes over and compiles the C to object code.

IMPLEMENTATION OF VIRTUAL FUNCTIONS

As noted, polymorphism is accomplished by using inheritance and virtual member functions. How does the C++ compiler decide which function should be called when an instance of unknown type is used? In the current release of MPW C++, every instance of an object in an inheritance hierarchy has a hidden data member, which is a pointer to a virtual function table. Each member function is known to be at a particular offset in the table. The member functions for the different classes in an inheritance chain are stored in different tables. The table pointed to is determined at the time of object creation. (See the sidebar called "Layout of Objects and Their Virtual Functions in Memory.")

So far, nothing in the implementation of virtual functions seems to preclude their use in nonapplication contexts. Once an object is instantiated, the code needed to call a virtual function can be executed from any context, including stand-alone code resources. However, MPW C++ does not currently support a mechanism to allocate storage for, or to initialize, the virtual function tables in nonapplication contexts.

CODE RESOURCE SUPPORT FOR POLYMORPHISM

As noted above, virtual function tables are required for polymorphism in C++. To support virtual function tables in stand-alone code, two issues must be resolved:

- How to allocate the virtual function tables.
- How to initialize the virtual function tables.

LAYOUT OF OBJECTS AND THEIR VIRTUAL FUNCTIONS IN MEMORY

For a typical class such as class **`foo`**, how does the compiler generate code to call the proper virtual function at run time? The following class and diagram show how this is accomplished.

```
class foo {
public:
 virtual void method1 ();
 virtual void method2();
private:
 int member1;
 int member2;
};
```

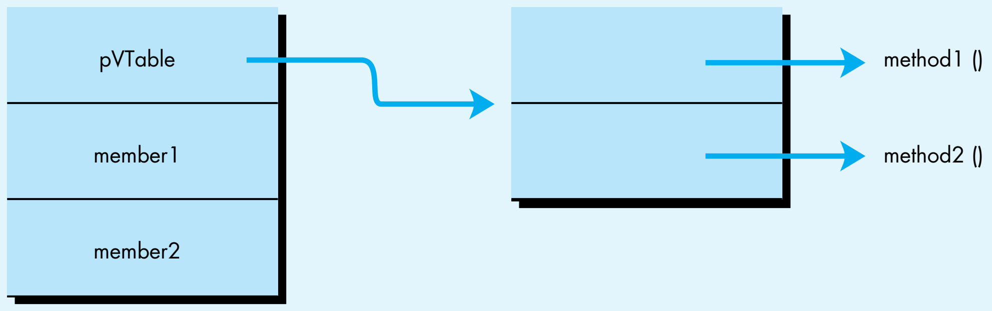


Figure 2
Calling Virtual Functions at Run Time

As shown by the figure, an instance of class **`foo`** has three data members. Two of the members, **`member1`** and **`member2`**, are part of the class definition, while a third member we'll call **`pVTable`** is a hidden member automatically created by the compiler. **`pVTable`** is a pointer to a table of function pointers (also automatically generated by the compiler) that holds pointers to all the functions in the class that are declared virtual. The code that is generated to call a virtual function is therefore something like this:

```
// Code written in C++:
myFoo->method1();
/* Becomes this code in C: */
(*myFoo->pVTable[0])();
```

This is the memory layout for a virtual function table used in single inheritance. For multiple inheritance, the structures used are more complicated.

GLOBAL VARIABLES IN CODE RESOURCES

In MPW C++, virtual function tables live in C global variable space. Unfortunately, the MPW languages do not support the use of global variables in stand-alone code. However, Technical Note #256, Stand-Alone Code, *ad nauseam*, shows how to add support for global variables in standalone code resources. In simple terms, this involves allocating storage for the globals, initializing the globals, and arranging for the proper value to be placed in machine register A5. These functions can be neatly expressed as a class in C++. The following class, called **`A5World`**, provides these services.

```
class A5World {

public:
  A5World();        // Constructor sets up world.
  ~A5World();       // Destructor destroys it.

  // Main functions:  Enter(), Leave().
  void Enter();     // Go into our world.
  void Leave();     // Restore old A5 context.

  // Error reporting.
  OSErr Error()     { return error; }

private:
  OSErr error;// The last error that occurred.
  long worldSize;   // How big our globals are.
  Ptr ourA5;        // The storage for the globals.
  Ptr oldA5;        // Old A5.
};
```

To use globals, a code resource written in C++ merely creates an instance of an **`A5World`** object. Here is an example:

```
// Hello_A5World.cp
// Simple code resource that uses global variables.

#include "A5World.h"

// Array of characters in a global.
char global_string[256];
```

```
void main()
{
      // Temporarily create global space.
      A5World ourWorld;

      // Check for errors.
      if(ourWorld.Error() != noErr)
            return;

      // We got it; let's go inside our global space.
      ourWorld.Enter();

      // Use our global variable.
      strcpy(global_string, "Hi there!");
      debugstr(global_string);

      // Time to go home now.
      ourWorld.Leave();

      // The destructor automatically deallocates
      // our global space.
}
```

The full implementation of class **`A5World`** appears on the *Developer Essentials* disc (Poly. in Code Resources folder). By itself, this is a useful piece of code.

INITIALIZING THE VIRTUAL FUNCTION TABLES

As noted, MPW C++ is implemented as a language translator (called CFront) that translates C++ to C. As you might guess, classes are implemented as structs in C, and member functions are just ordinary C functions. As also noted, the virtual function tables are implemented as global variables. We have solved the problem of having globals in stand-alone code, so the remaining issue is how to initialize these tables with the proper pointers to the member functions.

The initialization of a global variable with a pointer to a function is not supported in stand-alone code written in MPW languages. This initialization is normally done by the linker, which creates a jump table, and the current version of the MPW Linker will not generate jump tables for stand-alone code. Therefore, the only way to initialize global variables with pointers to code is manually at run time.

To understand the solution to this problem, let's take a look at what CFront does when it sees a hierarchy of classes with virtual functions. Here is a simple hierarchy of two classes, **`Base`** and **`Derived`**:

```
class Base {
public:
      Base();
      virtual void Method();
};

class Derived : public Base {
public:
      Derived();
      virtual void Method();
};
```

When MPW C++ sees these class definitions, it emits the following C to allocate and initialize the virtual function tables:

```
struct __mptr __vtbl__7Derived[]={0,0,0,
0,0,(__vptp)Method__7DerivedFv,0,0,0};
struct __mptr *__ptbl__7Derived=__vtbl__7Derived;
struct __mptr __vtbl__4Base[]={0,0,0,
0,0,(__vptp)Method__4BaseFv,0,0,0};
struct __mptr *__ptbl__4Base=__vtbl__4Base;
```

The variables **`__vtbl__4Base[]`** and **`__vtbl__7Derived[]`** are the virtual function tables for the classes `Base` and `Derived`. To support polymorphism in stand-alone code, this code must be split into two parts: a declaration part and an initialization part. The initialization part is simply a C function that initializes the tables. The following code shows how the tables might be transformed for use in stand-alone code:

```
struct __mptr __vtbl__7Derived[3];
struct __mptr *__ptbl__7Derived;
struct __mptr __vtbl__4Base[3];
struct __mptr *__ptbl__4Base;

void init_vtbls(void)
{
      __vtbl__7Derived[1].d = 0;
      __vtbl__7Derived[1].i = 0;
      __vtbl__7Derived[1].f = (__vptp)Method__7DerivedFv;
      __ptbl__7Derived=__vtbl__7Derived;

      __vtbl__4Base[1].d = 0;
      __vtbl__4Base[1].i = 0;
      __vtbl__4Base[1].f = (__vptp)Method__4BaseFv;
      __ptbl__4Base=__vtbl__4Base;
}
```

What we end up with is a declaration of global variables and a simple C function that must be called before the virtual functions are called. The code transformation shown is easy to implement. The *Developer Essentials* disc contains a simple MPW Shell script and two MPW tools that perform this function. The script is called **`ProcessVTables`**, and the tools are called **`FixTables`** and **`FilterTables`**.

COMBINING THE CONCEPTS

What remains is to combine the concepts of allocating global variables and initializing virtual function tables into a single construct. The following code, class **`VirtualWorld`**, based on class **`A5World`**, provides these two services.

```
class VirtualWorld : public Relocatable {
public:
      // Constructor sets up world.
      VirtualWorld(Boolean worldFloats);
      // Destructor destroys it.
      ~VirtualWorld();

      // Main functions; Enter sets A5 to point to
      // our world.

      // Go into our world.
      void Enter();
      // Restore old A5 context.
      void Leave();

      // Error reporting.
      OSErr Result()    { return error; }

private:
      // The last error that occurred.
      OSErr error;
      // Whether we have to call the vtable init.
      Boolean codeFloats;
      // How big our globals are.
      long worldSize;
```

```
  // The storage for the virtual world.
  Ptr ourA5;
  // Old A5.
  Ptr oldA5;
};
```

The constructor for class **VirtualWorld** requires one parameter, **worldFloats**, a Boolean value that tells whether or not the code resource floats between calls. This flag is used to decide whether or not the virtual function tables need reinitializing on every call to the code resource. Code resources such as **WDEF**s do float, and can even be purged, so this flag is essential. If **worldFloats** is false, the virtual function tables are initialized once in the constructor. This initialization is performed by calling the function **init_vtables()**, shown earlier.

The **Enter()** and **Leave()** member functions set up and restore the A5 global space, respectively. If the member variable **codeFloats** is true, **Enter()** calls the **init_vtables()** each time.

As in the **A5World** class, the **Error()** member function reports error conditions, which should be checked before assuming the world is set up correctly.

HANDLE-BASED CLASSES

The C++ default storage strategy is to create objects as pointers. As we all know, using pointers to allocate storage on the Macintosh makes memory management a lot less efficient. The ability to store data in relocatable blocks allows the Macintosh to use more of its memory since relocatable blocks can be shuffled around to make space.

Luckily, Apple has extended C++ in a way that allows us to take advantage of the Macintosh Memory Manager by adding the built-in class HandleObject. The only restrictions placed on handle-based objects is that they can be used only for single-inheritance hierarchies. Most object programming tasks, however, can be handled using single inheritance.

To make handle-based objects easier to work with, here is class Relocatable, a class derived from HandleObject. Class Relocatable provides functions for manipulating handle-based objects without the hassle of all those casts.

```
class Relocatable : HandleObject {
protected:
    void Lock() {HLock((Handle)this);}
    void Unlock() {HUnlock((Handle)this);}
    void MoveHigh() {
       MoveHHi((Handle)this);}
    SignedByte GetState() {
       return HGetState((Handle)this);}
    void SetState(SignedByte flags) {
       HSetState((Handle)this, flags);}
};
```

EXAMPLE: AN ICONIFIABLE WINDOW DEFINITION

To show off this really cool technique, I have written one of everybody's favorite code resources, a window definition function, or **`WDEF`**, that uses polymorphism. The example demonstrates how to define a base class for windows that is easy to inherit from—so you can add a feature to a window while leaving the original window code untouched.

CLASS WINDOWDEFINITION

Class **`WindowDefinition`** forms the template for all other window definitions. Here is its interface:

```
class WindowDefinition : public Relocatable {
public:
      // Initialize window.
      virtual void New(WindowPeek theWindow)
            { itsWindow = theWindow; }
      // Destroy window.
      virtual void Dispose() {}

      // Compute all relevant regions.
      virtual void CalcRgns() {}
      // Draw the frame of the window.
      virtual void DrawFrame() {}
      // Draw the goaway box (toggle state).
      virtual void DrawGoAwayBox() {}
      // Draw window's grow icon.
      virtual void DrawGIcon() {}
      // Draw grow image of window.
      virtual void DrawGrowImage(Rect& growRect) {}
      // Do hit testing.
      virtual long Hit(Point& whereHit)
      { return wNoHit; }
protected:
      // Window we are keeping track of.
      WindowPeek itsWindow;
};
```

`WindowDefinition` uses methods to respond to all the messages to which a **`WDEF`** is expected to respond. All the methods are just placeholders here, as **`WindowDefinition`** is an *abstract* base class.

Class **`WindowDefinition`**'s superclass, **`Relocatable`**, provides services to all handle-based classes, such as locking **`this`**, moving it high, and unlocking it. This class makes the casts to type **`Handle`** that are normally necessary and makes dealing with handle-based classes pleasant—and safer.

CLASS WINDOWFRAME

The next class to look at is **`WindowFrame`**. **`WindowFrame`** implements a basic window that can be resized, moved, and shown in highlighted or unhighlighted state.

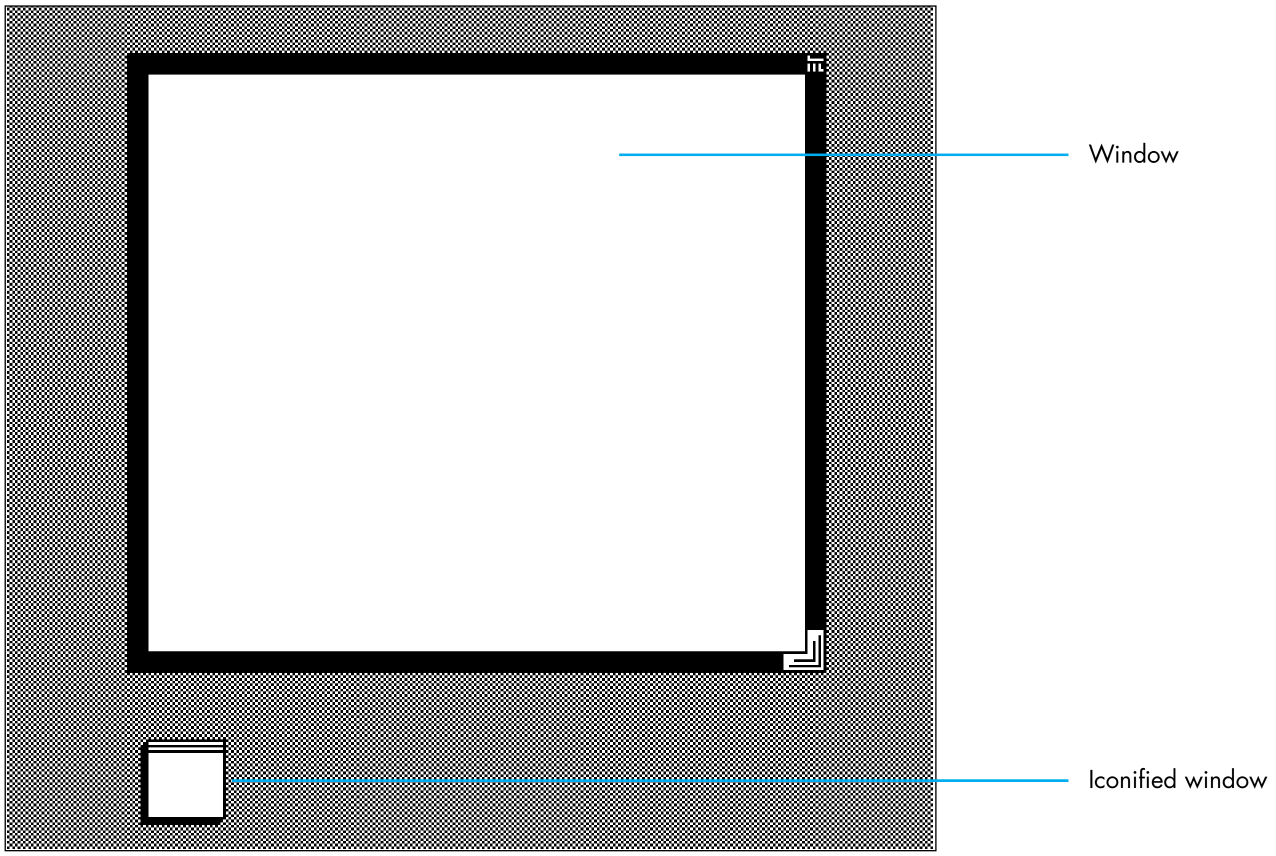


Figure 3
Class WindowFrame's Window on the Desktop

```
class WindowFrame : public WindowDefinition {
public:
      virtual void New(WindowPeek theWindow);
      virtual void Dispose();
      virtual void CalcRgns();
      virtual void DrawFrame();
      virtual void DrawGrowImage(Rect& growRect);
      virtual long Hit(Point& whereHit);
private:
      // Border between content and structure
      // boundaries.
      RgnHandle itsBorderRgn;
};
```

The code that makes this window work is included in the full source example on the *Developer Essentials* disc.

CLASS ICONWDEF

To implement a window that is iconifiable, we can derive from the class **`WindowFrame`**, and modify its behavior, without having to rewrite the code that implements the window. All an icon has to do is respond to clicks and be dragged around. So, the class **`IconWDef`** just has to worry about keeping track of whether the window is iconified or not, and lets the **`WindowFrame`** part take care of being a window. Here is the interface to class **`IconWDef`**.

```
class IconWindowDef : public WindowFrame {
public:
      // Window methods.
      virtual void New(WindowPeek theWindow);
      // We have different regions when iconified.
      virtual void CalcRgns();
      // We draw an icon if in the iconified state.
      virtual void DrawFrame();
      virtual long Hit(Point& whereHit);
private:
      // State of our window.
      Boolean iconified;
      // If we've ever been iconified.
      Boolean everIconified;
      // Flag that says we want to change our state.
      Boolean requestingStateChange;
      // How many times CalcRgns has been called.
      short calcRgnsCount;
      // Place to hit to iconify window.
      Rect iconifyRect;
      // Where to put when iconified.
      Point iconifiedLocation;
};
```

The decision about when to iconify the window is made in the **`IconWDef`**'s **`Hit()`** method. In the current implementation, if the window's zoom box is hit with the Option key held down, the window toggles between being an icon and being a window.

SUMMARY

This article has shown how to use polymorphism—the combination of inheritance and run-time binding of functions to objects—in the context of stand-alone code resources. Issues that had to be resolved were how to provide support for globals in stand-alone code and how to arrange for the initialization of virtual function tables.

Although the code for the example shows how to use polymorphism for window definition functions, you can use the same technique to write any type of code resource: menu definitions, list definitions, control definitions, and even drivers.

ROOM FOR IMPROVEMENT

The code could be improved in two ways:

- By using handles to allocate the A5 globals and passing in a parameter to tell `VirtualWorld` that the data can float.
- By removing the QuickDraw-specific code and placing it in a subclass of `VirtualWorld`.

CAVEATS

A couple of words of warning are in order. The tools that process the virtual function tables depend on the way CFront generates the tables. If AT&T or Apple ever decides to change the way these tables are generated (probably unlikely), the tools described in the example will probably break. However, it would not be difficult to modify the tools if changes were made.

Classes inherited from class `PascalObject` are not supported by the techniques described in this article. This is because these classes do not implement run-time binding using virtual function tables. This is not a problem since `PascalObject`s were intended only for use with MacApp, and (for now) MacApp can be used only for applications.

Look at the code on *Developer Essentials* for more information, and good luck!

Thanks to Our Technical Reviewers
Herman Camarena, Jack Palevich •

SYSTEM 7.0 SNEAKS

For those who missed the Q & A session on System 7.0 at the May 1990 Apple Worldwide Developers' Conference, here's a sampling of what went on.

Q

It would be nice to have Apple Events that could get a list of valid Apple Events from an application. Is that going to be available?

A

There are Apple Events to do that kind of thing already, and if you discussed the issue on AppleLink® you'd find that out right away. *That's a hint. Get it?—ED.*

Q

Can I drop Control Panel device files into the Apple menu folder and launch them directly from the Apple menu?

A

The short answer is yes. The more complete answer is this: Anything you could double-click from the Finder™ can be dropped into the Apple menu folder and launched from the Apple menu.

Q

My corporation has billions and billions of zones. We need a larger zone-name selection area in the Chooser. A list with the first few characters isn't good enough. Can this be fixed?

A

This is one of those things that used to be "an important future direction," so we did something about it. And now it's fixed in System 7.0.

Q

In many cases, you have the same application in different versions with the same creator. It would be nice if you could double-click a document and consistently turn on the oldest version, or the latest version, instead of just the one you last copied. Is this possible?

A

Not only is that possible, but—except for what looks like a small bug—it's already implemented and in the version of System 7.0 you have.

Q

You said earlier that a driver would lock down a completion routine. I just wondered how a piece of software could lock down code that's of unknown extent and possibly discontiguous.

A

You must have misunderstood. The Device Manager takes care of holding down the parameter block and also takes care of not calling the completion routine until paging is safe, so the completion routine itself doesn't have to be held down. Neither does any of the data it touches. The driver doesn't have to do anything. It's all handled by the Device Manager.

Q

But how does the Device Manager know that the completion routine is going to be around?

A

It doesn't. The Device Manager defers calling the completion routine until paging is safe.

Q

What limit is there, if any, on the number of published and subscribed items allowed for an application?

A

For a particular document, the maximum number is the maximum number of open files available, and, for an application, the maximum is limited by disk space.

Q

Are there changes or do you have plans for changes to the way applications are allocated memory under the omnipresent MultiFinder®? For instance, is there a way to dynamically grow their memory? I find that I have sort of a "normal" size for an application. And then, once in a while, I switch to Finder to give the application the entire memory.

A

If you're developing an application, you can use temporary memory and eliminate most of the need to do this. We're certainly investigating models that won't require the somewhat stilted setting of partition size that we have today.

Q

Dear Miss Manners: Will the operating system have the capability of launching a hidden application? An example is when you want to have one application launch another, send it some high-level events to potentially select some text, copy to the Clipboard, and then quit the application, without having the application show up and do all that in front of the user.

A

At first glance, this looks like an application-level thing, and a product could be designed this way. We're not planning to support this type of thing directly in the OS.

Q

Would it be feasible to keep Help resources in a separate file? If opened from within the application, these resources should appear in the regular Resource Search Path.

A

That's absolutely right. If you keep the file open and in the path when you call the Help Manager, you can keep the Help in a separate file.

Q

Is there a lot of authentication for high-level events, or should applications that need password protection implement the events themselves through the PPC toolbox in high-level events?

A

We have authentication built into all the interapplication communication stuff, through the Users and Groups portion of the Control Panel.

Q

This one's addressed to the Macintosh gods and goddesses. Using high-level events, is there a mechanism for bringing a background application to the front? If not, is there any way for the application to bring itself to the front?

A

The simple answer is, the Process Manager now has calls for doing this. In terms of user interface, though, it's probably a very bad idea to spring an application on the user from the background. He or she could be initializing a disk, or something!

Q

What is the Apple definition of 32-bit clean?

A

This is one of our favorite questions. What *32-bit clean* refers to is an application that doesn't contain anything to prevent it from running on a system with a 32-bit memory manager. The application can't contain any code that relies specifically on 24-bit stuff. Examples are masking the high bit, treating addresses as signed addresses, or using some specific old calls in low-memory globals whose accessibility is intrinsically limited to 24-bit addressing. The idea is explained in a lot of detail in Technical Note #212, so that's the Apple definition of *32-bit clean:* Tech Note #212.

Q

Will System 7.0 run on a Macintosh 512K enhanced ("512Ke") with third-party memory, a SCSI upgrade, and 2 MB of memory?

A

We hope so! There are thousands of us who have those! And our plan is that it'll work on those configurations. There's nothing we're doing to prevent that.

Q

Can a Macintosh boot off the network?

A

Current Macintosh computers don't support network booting (that is, booting off a file server), mostly because the ROM code doesn't know to look out on the network for boot devices. In the future, you'll see network-bootable Macintoshes, though. It's going to take a little bit of time, but you'll see those.

Q

Can a user lock out his or her disk from network access, high-level events, and so on, for a set time (during data acquisition, for example)?

A

There is a user interface to turn off all Apple Events, but not for a set time. You turn it on and off in the network setup Control Panel. And there's a programmatic interface to shut down file sharing if it's currently running.

Q

Is FileShare simply a personal AppleShare® server? Could you please compare and contrast the capabilities of AppleShare and FileShare?

A

The guts of FileShare is really a personal server. FileShare provides all the same call support as any AppleShare server, so the inner workings are the same. However, a lot of the tuning, a lot of the performance, and a lot of the upper limits have been lowered. FileShare is intended only for two or three people. It's much simpler, and provides more personal services, than the big, full-blown dedicated server.

Q

If System 7.0 has a real font size and width—for example, if the width of the letter is 8.5 pixels—is the width rounded to 8?

A

Font sizing is the same as before System 7.0. It's affected by the Set Fract Enable call in the Font Manager, and there's also a fractional pen position in color QuickDraw, so that the rounding works properly over multiple characters, rather than over a single character. This hasn't changed.

Q

Is the Database Access Manager intended to provide a vendor-independent interface for database packages?

A

The Database Access Manager is really intended to provide a vendor-independent interface to data. You could use it to talk to Macintosh databases, or whatever.

Q

Whatever happened to the Foreign File System Manager? Last year, we talked about being able to use ProDOS and MS-DOS volumes on the desktop.

A

The unfortunate answer is that the Foreign File System Manager didn't make our schedule for System 7.0, but we're working on it for a subsequent release.

MACINTOSH Q & A

Q

I am confused about the service routines and data areas passed in the _ADBOp call. What does it all mean?

A

That 's a good question. The ADBOp call looks like this:

```
FUNCTION ADBOp (data:Ptr;
compRout:ProcPtr;
buffer:Ptr;
commandNum:INTEGER) :
oserr;
```

data is a pointer to the "optional data area." This area is provided for the use of the service routine (if needed).

compRout is a pointer to the completion or service routine to be called when the _ADBOp command has been completed. It has the same meaning as the service routine passed to the _SetADBInfo call.

buffer is a pointer to a Pascal string, which may contain 0 to 8 bytes of information. These are the 2 to 8 bytes that a particular register of an ADB device is capable of sending and receiving.

commandNum is an integer that describes the command to be sent over the bus.

There is some confusion over the way the completion routines are called from _ADBOp. You can call these routines in one of three ways, depending on what you want to do:

If you do not wish to have a completion routine called, as in a Listen command, pass a NIL pointer to _ADBOp.

If you wish to call the routine already in use by the system for that address (as installed by _SetADBInfo), call _GetADBInfo before calling _ADBOp, and pass the routine pointer returned by _GetADBInfo to _ADBOp.

If you wish to provide your own completion routine and data area for the _ADBOp call, simply pass your own pointers to the _ADBOp call.

Remember, there should rarely be a reason to call _ADBOp. Most cases are handled by the system's polling and service request mechanism. In the cases where you must call _ADBOp, don't do it in a polling fashion, but as a mechanism for telling the device something (for example, telling the device to change modes or, in the case of the extended keyboard, to turn an LED on or off).

Q

The AppleTalk spec claims a data rate of 230.4 kbaud, which should require a 3.6864 MHz input to the SCC, but RTxCB on the Macintosh carries a 3.672 MHz clock. How does the AppleTalk driver reconcile this discrepancy and what frequency should I use?

A

The SCC contains a phase-locked loop that can lock on and synchronize with AppleTalk transmissions whose clock

These questions and answers are compiled by the Macintosh Developer Technical Support group. •

rates are not exactly to specifications, so everything is fine as long as both ends of the communication are using approximately the same clock frequency. If you are designing your own AppleTalk hardware from scratch, it's easiest to use a 3.6864 MHz oscillator and a Z8530. This has been tested and works just fine.

Q

When I fill in the fields of MPW's Name Binding Protocol (NBP) EntityName structure, AppleTalk doesn't recognize the entity, even though I know it's out there. What's going on?

A

The real definition of EntityName is three PACKED strings of any length (32 is just an example). No offsets for Asm are specified since each string address must be calculated by adding the length byte to the last string ptr. In Pascal, string(32) will be 34 bytes long (fields never start on an odd byte unless they are only 1 byte long). So correct-looking interfaces for Pascal and C will be generated, but they won't be the same, which is OK since they aren't used.

The point here is that you should never try to access the fields of the EntityName field directly. The only reason the type is defined at all is so that you can allocate EntityName variables that will hold the largest possible EntityName. To fill in an EntityName record, you should call the NBPSetEntity routine.

Q

How do I determine which language is in use on the system?

A

Every language has a corresponding KCHR resource. *Inside Macintosh*, volume I, page 499, lists the currently defined country codes, which are the resource IDs of the KCHR resources.

To find out which KCHR is in use, call the Script Manager function GetScript with the verb smScriptKeys. This call returns the ID of the KCHR resource in use (not the ID of the KEYC resource, as stated in *Inside Macintosh*, volume V, page 312).

Here's a bit of C code that determines which KCHR is being used:

```
#include <script.h>
 ...
kchrID = GetScript(smRoman,
smScriptKeys);
```

kchrID will be 1 when booted in French, 2 when booted in British English, and so on.

Q

When I use DeleteRevision to remove old revisions from my Projector database, the actual size of the ProjectorDB file doesn't decrease much. How can I make the file smaller?

A

Projector does not currently compact files. What it does is mark the areas of the database that are now free and put them into a free page list. This effectively puts holes into your database, holes that are subsequently filled up when you add more revisions.

Your database will get smaller only if the free pages are at the end of the file; then Projector will shrink the file. However, there is very little you can do about controlling this situation. If you absolutely must have a smaller database, then all you can do is check everything out, orphan the files, and create a new database. The disadvantage of this method is that you lose all your revisions and revision comments.

The Projector team is aware of the need to compact the database. The team is currently studying the feasibility of adding such a function.

Q

How does MultiFinder decide the starting order when you set multiple applications to start up under MultiFinder with Set Startup? Is there any way to control the order?

A

Here's the lowdown on MultiFinder application startup procedures.

From the Finder, launch, in order, application A, application B, and then application C. Switch to the Finder, choose Set Startup, and select Open Applications and DAs. The launch order is now application C, then application B, then application A. Regardless of the type of view from the Finder, the startup order is from top to bottom, respectively.

If you're *really* interested, the Finder Startup file in the System Folder contains the applications and files to be launched and the order in which they should be launched. This file contains a 'fndr' ID = 0 resource that stores the applications and their pathnames. The applications are launched in the order in which they are listed. You can use ResEdit to view the resource and see the filenames, the VRefNums, and the volume names of the startup applications. You can also tell the number of startup applications by the number at the beginning of the resource (that is, 0000 0001 means one item).

Remember, however, that this information is valid only for pre–System 7.0 MultiFinder environments.

DEVELOPER ESSENTIALS: ISSUE 4

Scott Converse, Corey Vian, Cleo Huggins, and Mary Skinner put develop in electronic form. Read more about the Electronic Media Group below.

The allegedly 27-year-old Jack Hodgson, product manager of *Developer Essentials*, produced and directed corporate videos in Boston, ran a small computer-book publishing company, did some free-lance programming, and founded the Boston Computer Society's Mac Users Group. His next big life goals are to buy his own plane and to learn to play his piano well enough to cut loose in Dave Szetela's Excellent Annual WWDC Moofamania Jam Sessions (caution: unofficial title).

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.

SCOTT CONVERSE is the group's Electronic Media Mogul and leader. A true on-line addict, he makes a living cruising the electronic highways and getting information to as many people as possible by using computers. Scott also loves sci-fi (particularly cyberpunk), reads books on design, and plays music on any of six full-blown, wall-shaking stereo systems in his house. When not cruising the electronic highways, he's racing radio-controlled cars. Would you ride the fiber optic byways with this guy? •

COREY VIAN takes the Zen approach to most things. He has an interdisciplinary B.A. in art and math from Maharishi International University. (Really! It's in Iowa.) An eleven-month Apple veteran (two years and eleven months if you count his prior consulting), he's now doing information interface design. An avid meditation practitioner, he also flies airplanes, builds cabinetry, windsurfs, snow skis, practices aikido, and composes R&R music—and he claims he isn't busy. •

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.

International System Software
Developer Essentials includes all the latest international versions of Macintosh system software as well as the latest U.S. versions of GS/OS and ProDOS, all in DiskCopy image format. (You must have a Macintosh to run DiskCopy and create floppy disks from these images.)

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.

DTS Technical Notes and Sample Code
All Apple II and Macintosh Technical Notes and Sample Code programs are included for your reference. Be sure to check here for the latest and greatest development information and Developer Technical Support programming tips and techniques.

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.

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.

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.

CLEO HUGGINS studied graphic design at the Rhode Island School of Design, taught design and semiotics at the Portland School of Art in Maine, and created the music typeface "Sonata" when she worked at Adobe. She received an M.S. in digital typography from Stanford University, and plays electric violin. Cleo always knew the computer would be a good place to combine her interests; she joined Apple to help refine the use of typography and design (and maybe even music) in our CDs.

MARY SKINNER collects the input, supervises testing, processes the feedback, and is the group's systems administrator (thank goodness Mary is a HyperCard fanatic). She's a native Iowan born in New York City. Her B.A. degrees in physics and Russian from the University of Iowa landed her as an Air Force lieutenant at Johnson Space Center from 1980 to 1984. Now an independent consultant, she likes to play with the computer, read sci-fi, and listen to the nonsoft side of rock and roll.

APPLE II Q & A

Q

How can I force text-page-two shadowing on the Apple IIGS?

A

Most uses for text-page-two shadowing come from older, 8-bit applications that use text page two. On the Apple IIGS, a Monitor ROM routine at $F962 (TEXT2COPY) toggles shadowing of text page two, through hardware on ROM 3 and through software on older machines. (A heartbeat task copies the bank $00 screen to the bank $E1 screen for software shadowing.)

TEXT2COPY is only a toggle—it can't tell you the current state of shadowing. To see if shadowing is currently enabled (the user may have enabled it manually with the Alternate Display Mode desk accessory), try storing a character in the bank $00 text-page-two screen, waiting more than 1/60th of a second and seeing if the character has been copied to bank $E1.

Q

Some of the toolbox calls I make crash when executed with GSBug active, but behave normally when GSBug isn't present. How come?

A

GSBug is intolerant of toolbox calls made in 8-bit mode. Although the *Apple IIGS Toolbox Reference* (pages 1–2) clearly states that all toolbox calls must be made in full native mode, the current tool dispatcher protects you by beginning with a REP #$30 instruction. GSBug does not. Be sure to make all toolbox calls in full native mode.

Q

If I try to select a file in an SFPutFile dialog box and the file already exists, clicking Save produces no action if I've entered ProDOS 8 since rebooting. Why?

A

The System Software 5.0.2 Resource Manager does not restart correctly on return from ProDOS 8. It doesn't correctly add the system resource file into the search path. When Standard File detects that you're trying to save over an existing file, it calls ErrorWindow to display a dialog box with the warning, "That file already exists," and the choice to replace or cancel. ErrorWindow fails because the system resource file is not open and the AlertWindow template can't be loaded. Standard File treats an error in the ErrorWindow call as if you'd clicked Cancel in the "That file already exists" dialog box. The net effect is that nothing at all happens. This is corrected in System Software 5.0.3.

These questions and answers are compiled by the Apple II Developer Technical Support group. •

Q

Why do Apple IIGS fonts look tall and skinny, as if they were made out of rubber and stretched too far in one direction? They look OK when I print using the "vertical condensed" option.

A

Nearly all the Apple IIGS fonts were originally designed for other systems, usually the Macintosh. Font definitions for the Apple IIGS and other systems are nearly identical. Macintosh pixels are square; the width-to-height ratio of a pixel is 1:1. Apple IIGS pixels are much taller than they are wide (the ratio for Apple IIGS 640 mode is about 5:12). When a font designed for square pixels is displayed on a system with pixels of a different shape, the characters look stretched. This is what happens on the Apple IIGS.

Apple could have changed the font strike for a more pleasing look at Apple IIGS resolutions, but for legal reasons such a change would require renaming the fonts. Times wouldn't be Times anymore, Helvetica wouldn't be Helvetica, and so on. The fonts would look the same, but the names would have to be different. In the tradeoff between appearance and well recognized font names, Apple chose to keep the familiar names and font strikes.

To compensate for the stretched fonts, all of Apple's printer drivers include a "vertically condensed" printer option. Selecting this option causes the printer drivers to print with double the screen's vertical resolution. Doubling the vertical resolution effectively makes the pixel aspect ratio about 10:12, or 5:6, which is close enough to square that the fonts look the way we expect them to.

Some fonts are designed for the Apple IIGS aspect ratio of 5:12. Such fonts are identified in their font family numbers by having the high bit set.

INSIDE THE MACINTOSH COPROCESSOR PLATFORM AND A/ROSE

The Macintosh® Coprocessor Platform™ provides a foundation for connectivity products such as the Serial NB Card, the TokenTalk NB Card, and the Coax-Twinax Card. Its operating system is A/ROSE, the Apple Real-time Operating System Environment. This article introduces you to the Macintosh Coprocessor Platform and A/ROSE, and gives you a taste of what is involved in developing a connectivity product on this foundation.

JOSEPH MAURER

The Macintosh Coprocessor Platform and A/ROSE together provide a hardware and software foundation for developers who want to create NuBus™ add-on cards for the Macintosh II family of computers. The developer's guide that comes with the kit is a hefty 400-page tome. If you're curious about how NuBus cards are built but not curious enough to tackle the developer's guide, read on. This article gives you an overview of the origins of the Macintosh Coprocessor Platform, its architecture, and details of its real-time, multitasking, message-based operating system, A/ROSE. It shows you some A/ROSE code. And it shows you how to experiment with some A/ROSE applications included on the *Developer Essentials* disc.

HOW IT ALL BEGAN

When development of various networking and communications products for the Macintosh II started at Apple, around 1987, it became obvious that the Macintosh Operating System didn't meet these products' needs for processing power and operating system capability. After all, the Macintosh OS was designed for human interaction rather than for connectivity to mainframe computers. It is not real-time (interrupts can be disabled for longer than is acceptable for fast interrupt-driven input/output), and it aims to provide a pleasant and efficient graphic user interface, rather than processor-intensive I/O handling.

JOSEPH MAURER, an 18-month Apple veteran, studied mathematics and theoretical physics at universities in Munich and Nice. Since then he's led a varied but somewhat theoretical life, which has included being a ballet school piano player, bicycle racer (champion of lower Bavaria!), math researcher, mountain climber, university professor, and Apple European technical support and training guru (you can decide for yourself if that one's theoretical or not). All in all, Joseph is basically a man of numbers: he has one wife, two Macintosh computers, three bicycles, and four children. He says he wants more Macs and more racing bikes (the ones he has are "slowing down"), but refuses to comment on wanting more wives and/or children. •

The solution was to make an "intelligent" NuBus card, with its own 68000 processor, its own working space in RAM, and its own basic operating system services; and to design this card not only as a basis for Apple's own products, but also as a tool for NuBus expansion card developers. The result was the Macintosh Coprocessor Platform. Its operating system, A/ROSE, was designed to respond to the needs of connectivity products, complement the capabilities of the Mac OS, and yet be generic enough to become the foundation for a new breed of message-based, distributed software architectures. The work on A/ROSE started in August 1987, and the first version was operational by February 1988.

Today, developers can build on this platform in designing products for communications and networking, data acquisition, signal processing, or any other heavy-duty processing. Time-consuming and/or time-critical tasks can be offloaded from the main logic board to a dedicated processor on the NuBus card. This increases the overall computational speed, of course, and allows for faster response times in the foreground applications. Moreover, unlike the standard Mac OS, A/ROSE provides the real-time and multitasking capabilities required for handling multiple communications protocols.

Nevertheless, A/ROSE on a Macintosh Coprocessor Platform still depends on the Mac OS (and its limitations—see Technical Note #221) for transferring large amounts of data across the NuBus through a driver to a Macintosh application. This means that ample data buffering (and careful error handling) should be provided on the card if the project requires high-performance data transfers. As you'll see in the next section, the card provides plenty of room for large buffers.

THE MACINTOSH COPROCESSOR PLATFORM UP CLOSE

The most prominent feature of the Macintosh Coprocessor Platform card is all the empty space on it, inviting hardware developers to heat up their soldering irons and to put plenty of advanced hardware on it. A complete master-slave NuBus interface comes for free, implemented by means of two chunky Texas Instruments ASICs (application-specific integrated circuits), 2441 and 2425. This interface manages to give the on-board MC68000 access to the whole 32-bit NuBus address space (by means of an address extension register). Conversely, the 24-bit address space of the local MC68000 can be accessed directly from across the NuBus. Custom hardware on the card can be enabled to take over the 68000 bus and even go to the NuBus, but A/ROSE tasks usually take care of servicing chips on the board, and communicate with the higher levels of the software design.

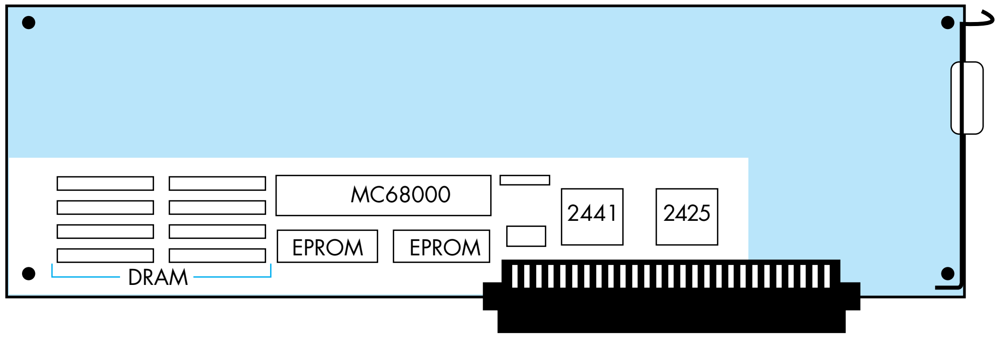


Figure 1
The Macintosh Coprocessor Platform Card

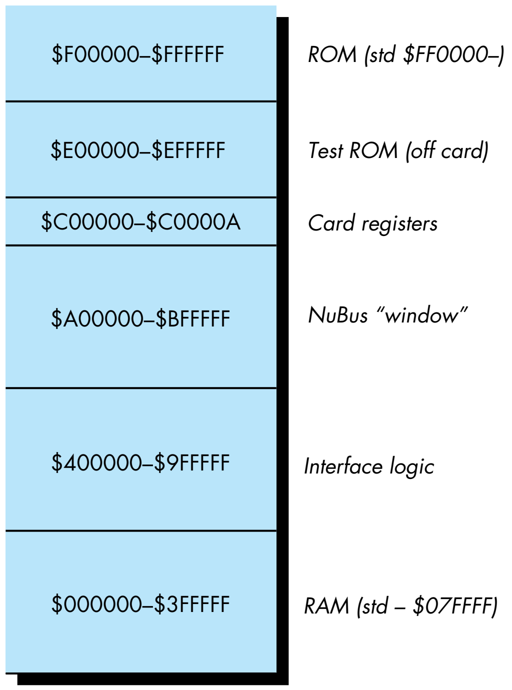


Figure 2
Memory Map of the Macintosh Coprocessor Platform With A/ROSE Running

The MC68000 on the Macintosh Coprocessor Platform card runs at 10 MHz (the NuBus clock speed) without wait states. Standard 512K of dynamic RAM is expandable up to 4M. Two 32K EPROMs contain the declaration ROM code needed to make the SlotManager happy, plus some pieces of code to help the MC68000 out of a Reset and to provide low-level diagnostic routines. The card also carries a programmable timer, used by A/ROSE for scheduling time-sliced tasks.

A/ROSE UP CLOSE

A/ROSE is a minimal, multitasking, distributed, message-based operating system. Here's what this means, in real terms:

It's minimal: The module that provides basic A/ROSE functionality, the A/ROSE kernel, fits into 6K; and a complete standard configuration of A/ROSE on a NuBus card amounts to only 23K of code and takes up only about 48K of buffer space. This leaves more than 400K for your code on a standard 512K RAM card. Still, as you will see, A/ROSE is a strong software platform to build on.

It's multitasking: A/ROSE does pre-emptive multitasking, with round-robin task scheduling (taking 32 priority levels into account).

It's real-time: A/ROSE offers 110 microseconds context switch time, with 20 microseconds of latency (guaranteed interrupt response time).

It's distributed and message-based: The A/ROSE software can be present on several cards, and it is completely autonomous and independent on each card. Tasks defined by users and by A/ROSE communicate with each other, even across the NuBus to other slots or the Mac® OS, by means of messages. These messages can carry pointers to data buffers along with them. Thousands of such messages can be passed per second (fastest from task to task within a card, and slower, of course, between different slots).

The A/ROSE kernel is responsible for task scheduling, interprocess communication, and memory management. The calls that correspond to these responsibilities are shown in Table 1. The standard configuration also includes utilities for bookkeeping and timer services. These utility functions are carried out by the A/ROSE managers: the Name Manager, the InterCard Communication Manager, the Remote System Manager, the Echo Manager, the Timer Library, the Trace Manager, and the Print Manager.

Table 1
The Ten A/ROSE Primitives

Name	Description
AROSEFreeMem()	Frees a block of memory[1]
FreeMsg()	Frees a message buffer[1]
AROSEGetMem()	Allocates a block of memory[1]
GetMsg()	Allocates a message buffer[1]
Receive()	Receives a message[2]
Reschedule()	Changes a task's scheduling mode
Send()	Sends a message[1]
Spl()	Sets the hardware priority level
StartTask()	Initiates a task
StopTask()	Stops a task

Notes:
1. Implemented in A/ROSE Prep with the same parameters.
2. Implemented in A/ROSE Prep with a supplementary parameter.

The A/ROSE architecture, shown in Figure 3, is completed by A/ROSE Prep, a version of A/ROSE that runs on the main CPU under the Macintosh Operating System and that is necessary to establish communication between the Mac OS and A/ROSE. The A/ROSE Prep file has the file type INIT, and contains among its numerous resources a DRVR named .IPC (for interprocess communication), and an INIT that executes at INIT31 time and basically installs and opens the .IPC driver. The .IPC driver takes care of the communication of Mac OS processes with A/ROSE tasks. Nothing can be downloaded to the Macintosh Coprocessor Platform if the A/ROSE Prep file is not in the System Folder: it contains card-dependent information needed for the download routines to succeed.

The programming interface to the .IPC driver (described in the A/ROSE header files arose.h, os.h, managers.h, iccmDefs.h, ipcGDefs.h, and provided through the library IPCGlue.o) mimics that of A/ROSE itself as closely as possible, providing the look and feel of A/ROSE even if there is no A/ROSE around. More practically speaking, with the A/ROSE Prep file in your System Folder, you can do a lot of interesting A/ROSE experiments even without a Macintosh Coprocessor Platform. For your convenience, the A/ROSE Prep file is included in the A/ROSE folder on the accompanying *Developer Essentials* disc.

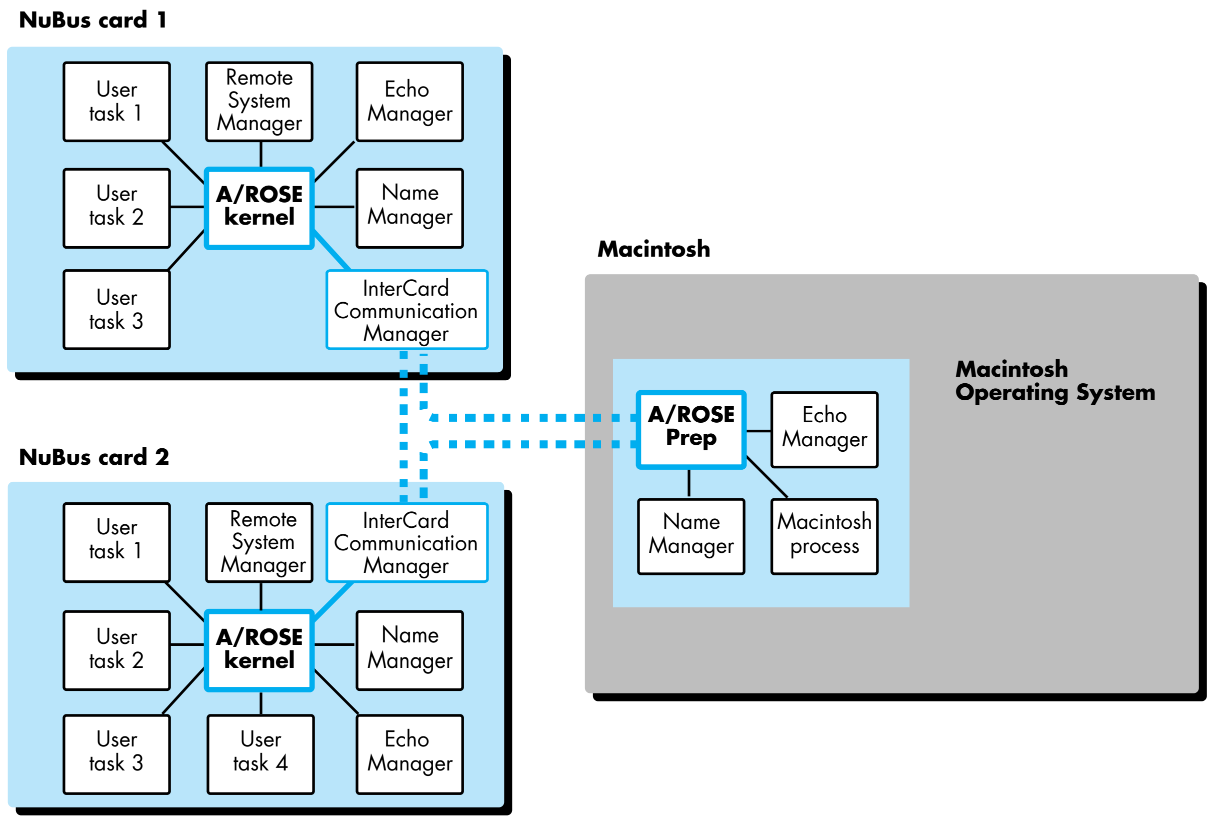


Figure 3
The Architecture of A/ROSE

WHAT'S THIS ABOUT MESSAGES AND TASKS?

Interprocess communication in A/ROSE takes place by means of messages passed back and forth between tasks. A typical example consists of a client/server relationship between A/ROSE program modules, as illustrated in Figure 4 on the next page.

The client task needs to know that the required server task exists; thus, the server task is initialized before the client task. Next, the client task issues a `GetMsg()` call to request a message buffer from a preallocated pool of message buffers that is maintained by A/ROSE and the size of which is specified by the user. After the message is filled with addressing information, command codes, and parameters, it is sent to the server task. At this point, the sending task loses rights to the message buffer, and should not use it again until it comes back through a `Receive()` call. On the other side, the server task usually sits in an infinite loop, waiting for messages requesting a service, handling these requests, and sending replies.

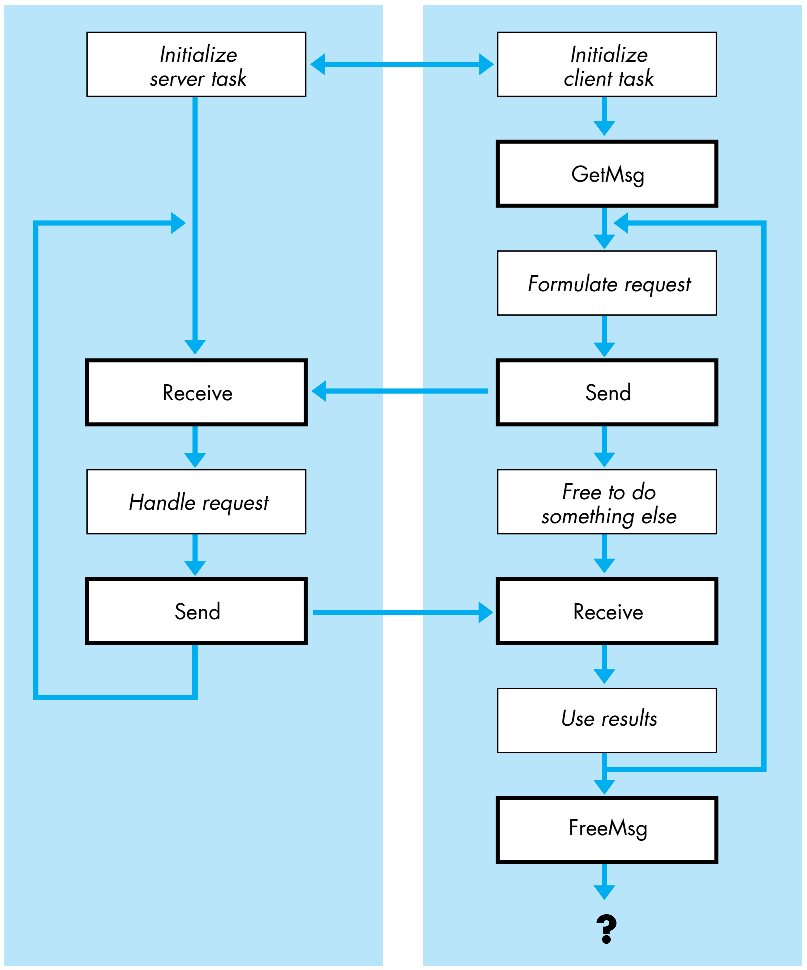


Figure 4
How Interprocess Communication Takes Place in A/ROSE

After receiving the reply, the client task can reuse the message buffer for subsequent requests, or release the buffer by means of a FreeMsg() call and go ahead with other business.

A message provides up to 24 bytes of user data, and is fixed length and asynchronous. If the data to be sent does not fit into the message proper, then it can be put anywhere in the sender's memory and the address and size of the data area can be passed in the message.

Each message is identified by a message ID and a message code. The message code is defined by agreement between the sender and the receiver. A convention followed in A/ROSE is for outgoing messages to use an even-numbered code and for replies to those messages to set the code to the next odd number.

The structure of an A/ROSE message is shown in Figure 5.

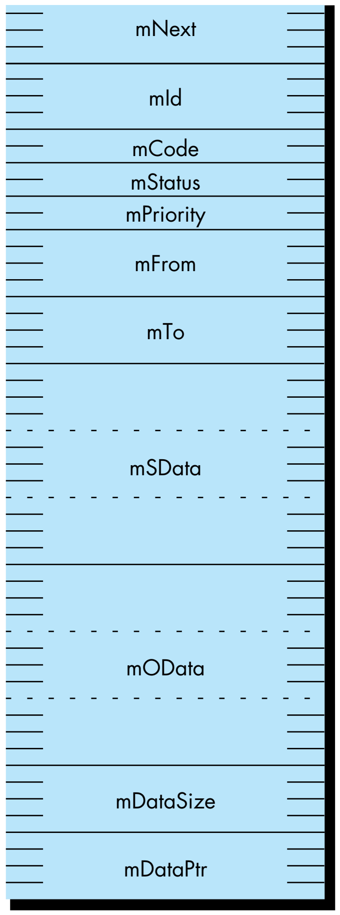


Figure 5
The Structure of an A/ROSE Message (54 Bytes)

In C, the messsage structure is declared as follows:

```
struct mMessage {
    struct mMessage    *mNext;     /* Used to chain messages internally.         */
    long               mId;        /* Unique identifier for a message.           */
    short              mCode;      /* User-defined message code.                 */
    short              mStatus;    /* Message return status.                     */
    unsigned short     mPriority;  /* Range is 0 (low) to 31 (high).             */
    tid_type           mFrom;      /* Task ID of task sending message.           */
    tid_type           mTo;        /* Task ID of task to which msg. is sent.     */
    unsigned long      mSData[3];  /* Used for sender's private information.     */
    unsigned long      mOData[3];  /* Used by receiver to send data back.        */
    long               mDataSize;  /* Size of data to which mDataPtr points.     */
    char               *mDataPtr;  /* Pointer to variable length data.           */
 };
```

Tasks in A/ROSE accept and reply to messages. A task is identified to A/ROSE by a task ID, which is a 32-bit field of type tid_type. Each task also has an associated name and type that is readable by humans. This is very close to the NameBinding protocol of AppleTalk in spirit and implementation.

Tasks are started with a call to the A/ROSE `StartTask()` primitive. Tasks have one of 32 priority levels, with level 31 as the highest priority and level 0 as the lowest. Tasks run either in slice mode or in run-to-block mode. In slice mode, a task runs for one major tick (about 50 milliseconds), and then relinquishes control of the CPU to a task of higher or equal priority, if one is available. In run-to-block mode, a task runs until it is blocked or until it completes. A task becomes blocked if it issues a `Receive()` call for a message that is not available. New tasks are scheduled for execution in the order of priority; a task is run only if no eligible tasks of higher priority are waiting.

TOKENTALK AND A/ROSE

by Anumele Raja

TokenTalk® is a typical application that runs under A/ROSE on an intelligent NuBus card. The following is a brief description of the TokenTalk hardware and how TokenTalk uses A/ROSE.

THE HARDWARE AND SOFTWARE

The TokenTalk NB card is the intelligent NuBus card that implements the Token Ring interface. The card consists of a 68000 processor and a Token Ring interface chip set made by Texas Instruments. The card's foundation is the Macintosh Coprocessor Platform. Besides TokenTalk, the card can also run MacAPPC™, MacDFT®, and MacSMB file transfer programs.

The Token Ring interface chip set is controlled by a program called Logical Link Control (LLC) that also implements the Token Ring protocol. LLC runs as a task under A/ROSE.

TokenTalk itself is an A/ROSE task that serves as the interface between programs running on the Macintosh and the LLC task. This task can be replaced by another task to implement other protocols like SNA.

THE DOWNLOAD PROCESS

When the user selects the TokenTalk device on the Network control panel, a resource file called TokenTalk Prep is loaded into the Macintosh and executed. TokenTalk Prep first finds a TokenTalk card, downloads A/ROSE if it is not already running on the card, and downloads the LLC task onto the card. The TokenTalk part of the AppleTalk device driver downloads the card part of the TokenTalk task by using TokenTalk Prep utilities, and starts the task.

Sound complicated? Let's take the operation sequence at a slower pace.

On the Macintosh side of TokenTalk, the operation sequence is as follows:

1. The user selects TokenTalk on the Network control panel.
2. The TokenTalk Prep file is loaded and started.
3. TokenTalk Prep makes sure that A/ROSE Prep is running on the Macintosh, and searches for a TokenTalk card by calling the `NewFindcard()` routine. If a TokenTalk card is found, TokenTalk Prep checks to see if A/ROSE is already running on that card by looking for a Name Manager. If A/ROSE is not running, TokenTalk Prep downloads onto the card a version of A/ROSE that includes the Name Manager, the InterCard Communication Manager, the Remote System Manager, and the Echo Manager. It then does a `Lookup_Task()` to find the LLC task. This task controls the Token Ring interface chip set and handles interrupts. If the LLC task is not found, it downloads the LLC task using the `DynamicDownload()` call.
4. The Network CDev then talks to the TokenTalk driver to activate TokenTalk. The TokenTalk driver, which resides on the Macintosh, is the interface between AppleTalk and the TokenTalk card. It operates by sending control commands and data to the TokenTalk task on the card and receiving status information and data.
5. The TokenTalk driver downloads the TokenTalk task to the NuBus card.

On the TokenTalk card side of TokenTalk, the operation sequence is as follows:

1. When the LLC task is started up, it registers itself with the object name LLC and the type name TokenTalk NB by calling the `Register_Task()` routine. It then calls the `Receive()` primitive and waits for messages. In the current implementation, the `Receive()` is issued with a timeout parameter. The LLC task runs under run-to-block mode.
2. When the TokenTalk task is started up, it registers itself with the object name Token Talk 1 and the type name TokenTalk NB by calling the `Register_Task()` routine. The TokenTalk task searches for the LLC task by doing a `Lookup_Task()`. It then waits for messages from the Macintosh to start an operation. When requests are received from the Macintosh, the TokenTalk task sends commands to the LLC task to carry out the various operations. The TokenTalk task does not control any hardware by itself.
3. Data is transferred from the Token Ring interface to the memory and vice versa by a direct memory access (DMA) mechanism built into the interface chip set. An interrupt is generated by the DMA device at the completion of a data transfer.

Both the LLC task and the TokenTalk task run with a priority of 30 and allocate a stack of 2048 bytes. No heap space is allocated by these tasks. TokenTalk Prep uses the start parameter block to pass information to the LLC task. This information specifies the TokenTalk address for the node.

THE A/ROSE MANAGERS

A manager in A/ROSE is just another task, which does its job in accepting and replying to messages with predefined message codes. As mentioned earlier, the A/ROSE managers are the Name Manager, the InterCard Communication Manager, the Remote System Manager, the Echo Manager, the Timer Library, the Trace Manager, and the Print Manager. The first four are discussed in greater detail here. Use of the Name Manager and the InterCard Communication Manager is demonstrated in the sample program ShowTasks and in "Building a Download File," later in this article.

THE NAME MANAGER

The Name Manager maintains a cross-reference between task IDs and their associated name and type. User tasks can register themselves with the Name Manager by specifying an object name and an object type, and then other tasks that need to refer to this task can look up the task by name and type by calling the A/ROSE `Lookup_Task()` utility. Conversely, for a given task ID, the Name Manager brings back the object name and object type if you send it a message with `mCode = NM_LOOKUP_NAME.`

The Name Manager also provides notification services. These services include signaling when a NuBus card is shut down or started up, checking to see if a task is present or not, and signaling when a task terminates.

THE INTERCARD COMMUNICATION MANAGER

The InterCard Communication Manager (ICCM) enables user tasks to communicate with tasks on other NuBus cards or on the main logic board. There are only three message codes a user task may send to the ICCM: `ICC_GETCARDS`, `ICC_DETACH`, and `ICC_ATTACH`.

`ICC_GETCARDS` returns a long integer for each of the sixteen possible NuBus slots. A positive number represents the task ID of the Name Manager running under A/ROSE on a Macintosh Coprocessor Platform card. For slot = 0, this is the task ID of the Name Manager incorporated in A/ROSE Prep, under the Macintosh OS. The task ID of a Name Manager is required to look up specific tasks on any card on the NuBus.

The message codes `ICC_DETACH` and `ICC_ATTACH` are provided for NuBus cards that get power from a source other than the NuBus, so that when the power to the Macintosh main logic board is turned off, the NuBus card continues to function. With these message codes, you can delink the NuBus card from the outside world, thus preventing access over the NuBus.

THE REMOTE SYSTEM MANAGER

The Remote System Manager running on a NuBus card enables tasks running on any other NuBus card or the main processor to execute certain A/ROSE primitives remotely. The A/ROSE primitives `A/ROSEGetMem()`, `A/ROSEFreeMem()`, `StartTask()`, and `StopTask()` are supported, enabling tasks to be downloaded, started, and stopped dynamically. The Remote System Manager registers itself with the Name Manager with the name RSM and the type RSM.

THE ECHO MANAGER

The Echo Manager echoes all messages sent to it. This can be very useful in the initial stages of testing A/ROSE applications.

DOWNLOADING TO THE CARD

All code running on a NuBus card is downloaded to the card's memory from the main logic board. Code can be downloaded statically or dynamically, to one or multiple cards.

In static downloading, the user builds the entire memory image of the application to be run on the card by linking the code with A/ROSE object files. The main program is user code; it calls `osinit()` to initialize A/ROSE and `osstart()` to start the operating system. Before starting the operating system, the main program must start all the necessary managers and user tasks. The memory image is downloaded onto the NuBus card using the static downloading facility, which halts the card, downloads the code, and starts the card again.

In dynamic downloading, the user downloads a generic version of A/ROSE onto a NuBus card by invoking `StartAROSE()`. Once the A/ROSE kernel and requisite managers are up and running, the user can download tasks using the dynamic downloading facility.

A/ROSE provides a number of ways to download the code. The MPW tool Download takes the pathname of a file as parameter, and tries to download it to every Macintosh Coprocessor Platform card it finds (if used without the optional slot parameter). This is convenient during the development cycle under MPW. Another possibility is to use the Macintosh application ndld. Finally, you can use the `NewDownload()` routine directly from within your own application. (See the sidebar on the next page for a description of Download and ndld.)

UTILITIES THAT MAKE IT EASIER TO DEVELOP A/ROSE PROGRAMS ON THE MACINTOSH

by Anumele Raja

The following Macintosh utilities, included on the A/ROSE distribution disks, facilitate development of A/ROSE programs on the Macintosh:

Print Manager (nprm) is a Macintosh application that enables users to display information from a task running on a Macintosh Coprocessor Platform card. It registers itself with A/ROSE by the object name Print Manager and the type name Print Manager. Strings to be printed are sent to the Print Manager by the printf routine supplied with the A/ROSE release. The first time printf is called, it looks for the Print Manager and finds its Task ID. Subsequently, it sends all print strings as messages to the Print Manager, which puts up a window and displays the strings it receives. Users can display diagnostic messages using printf. Print Manager features can also be implemented in a user's program.

Dumpcard is an MPW tool that dumps the status of A/ROSE tasks running on any NuBus card. Available options display the memory blocks, the messages waiting to be received by a task, and the task control blocks of all tasks running under A/ROSE. If the card stops for any reason, like a bus error, the user can get a trace of the stack to find the calling sequence that caused the exception. In addition, the user can request disassembly of instructions around the break point.

Download application (ndld) is a Macintosh application used to download A/ROSE and/or A/ROSE tasks onto a specified card or cards either statically or dynamically. The file selection is done through a standard Get File dialog box.

Download is an MPW tool that downloads A/ROSE and/or A/ROSE tasks onto a specified card or cards either statically or dynamically. It is useful when the user wishes to download code from a shell script.

NuBug is a debugging application used to debug A/ROSE programs running on a card. NuBug looks and works like MacsBug. All MacsBug commands that are not specific to the Mac OS are supported by NuBug. In addition, NuBug provides commands specific to A/ROSE, dealing with task status, task names, and such. NuBug is a multiwindow application that brings up as many windows as there are NuBus cards capable of running A/ROSE. Because NuBug is implemented in C++, it can be enhanced very easily.

Users can look forward to a new, we hope official, version of NuBug very soon. The current release of NuBug has not been tested formally and is not supported by Apple. Still, programmers find it so helpful that they don't seem to mind if they encounter a few glitches.

SOME SAMPLES OF A/ROSE PROGRAMMING

You'll find some samples of A/ROSE programming in the A/ROSE folder on the *Developer Essentials* disc. You can run these applications under MultiFinder after booting with A/ROSE Prep in the System Folder. With the exception of the downloading operation, all these applications will run whether or not your machine has a Macintosh Coprocessor Platform card installed.

You can take a closer look at the complete source code on the *Developer Essentials* disc. I'll show and discuss some fragments of it here.

TASKSAMPLE AND CLIENTAPPLI

The TaskSample application opens a window and waits for A/ROSE messages. The ClientAppli application looks for a server named myTaskName and sends a message on each button-click. The server TaskSample simply returns each message it receives to the sender, and ClientAppli displays the number of messages it has sent and received.

To experiment with producing alerts or error messages, launch both applications, then quit TaskSample and continue sending messages to it; restart it again and continue; or hit Command-Q immediately after the Send button, so that ClientAppli has gone by the time TaskSample sends the reply.

If you run the applications, you will notice a certain delay in messages being passed back and forth. This has to do with the SleepTime value (selected in the SleepTime menu), which is passed to the `WaitNextEvent()` call under MultiFinder. In the two sample programs, the A/ROSE Prep `Send()` and `Receive()` services are called only once at each tour through the event loop. Depending on the SleepTime value, the background application more or less slows down, and this explains the delay observed on the screen.

SHOWTASKS

ShowTasks is a tool that shows all the A/ROSE tasks that are "visible" in the machine (there might be "invisible" A/ROSE tasks, too). It goes through all sixteen NuBus slots, looks for all visible tasks, and displays them by task identifier, object name, and object type. Sample output of this program might look like the following (which reflects the situation where TaskSample and ClientAppli are running):

```
slot = $0 :
00000003: name "echo manager", type "echo manager"
00000004: name "myTaskName", type "myTaskType"
00000005: name "ClientApp", type "ClientType"
```

This indicates that there are no A/ROSE tasks running on a NuBus card at this time; slot $0 represents the good old main logic board where the A/ROSE Prep driver does its best to make us believe that there is an instance of A/ROSE.

Now let's look at some of the source code. For the sake of clarity in the following fragments, error handling is completely suppressed. Needless to say, nobody should ever try to compile this sort of code! The source code on the CD gives a more realistic idea of A/ROSE programming.

Here are the outlines of `main()` and the two basic subroutines `AskICCM()` and `NameLookup()`, with explanatory text following the code:

```
static tid_type cards[16];    // Place for 4 bytes per slot.
main()
{
      short slot, index;
      tid_type    tid;

      (void) OpenQueue(nil);  // Set up a message queue for me.
      AskICCM();  // Request Name Manager TID for each slot, store in cards[].
      for (slot=0; slot<16; slot++) {
            if (cards[slot] > 0) { // Name Manager TID is OK.
                  printf("\nSlot = $%X :\n",slot);
                  index = 0;
                  while (tid = Lookup_Task("=", "=", cards[slot], &index))
                  // Ask Name Manager for info about registered tasks.
                  NameLookup(cards[slot], tid);
            }
      }
      CloseQueue(); // Be nice with A/ROSE Prep.
} // End main().

void AskICCM()
{
      mMessage    *m;

      m = GetMsg();

      m->mTo       = GetICCTID();
      m->mCode     = ICC_GETCARDS;
      m->mDataPtr  = (char *) cards;
      m->mDataSize = sizeof (tid_type) * 16;
      Send(m);
      m = Receive(OS_MATCH_ALL, OS_MATCH_ALL, ICC_GETCARDS+1,
            OS_NO_TIMEOUT, 0);
      // SlotInfo is now in cards[0..15] (if nothing failed!).
      FreeMsg(m);
} // End AskICCM().
```

```
#define bufferSize 512
void NameLookup(tid_type ntid, tid_type tid)
// ntid = Name Manager TID.
// tid = ID of the task for which we request the name.
{
      struct pb_lookup_name   *lnam_ptr; // (See text.)
      char  buffer[bufferSize];
      mMessage    *m;

      m = GetMsg();
      m->mTo       = ntid;
      m->mCode     = NM_LOOKUP_NAME;
      m->mDataPtr  = buffer;
      m->mDataSize = bufferSize;
      lnam_ptr = (pb_lookup_name *) &buffer;
      lnam_ptr->lnm_index = 0;
      lnam_ptr->lnm_tid  = tid;
      lnam_ptr->lnm_RAsize = bufferSize -
                        (sizeof(pb_lookup_name) - sizeof(ra_lnm));
      Send(m);
      m = Receive (OS_MATCH_ALL, OS_MATCH_ALL, NM_LOOKUP_NAME+1,
                                                 OS_NO_TIMEOUT, nil);
      DisplayTaskInfo(lnam_ptr); // Lots of silly string handling.
      FreeMsg(m);
} // End NameLookup().
```

The `OpenQueue()` call is needed to make use of the A/ROSE Prep services; it takes a procedure pointer as parameter. If a procedure is specified, it gets called repeatedly during a blocking `Receive()` request, which avoids blocking the machine during waiting. In our case, we don't use blocking receives, and don't need this feature. By the way, `OpenQueue()` returns a task identifier that will be ours for the rest of the process.

We have to deal with the InterCard Communication Manager, in `AskICCM()`, and the Name Manager, indirectly in `Lookup_Task()` and directly in `NameLookup()`. First, we want to ask the InterCard Communication Manager what it knows about the sixteen NuBus slots. Naturally, we send a message.

The local variable `m` is declared as a pointer to the `struct mMessage` (note the spelling, in order to distinguish it from the `message` field in an EventRecord). The `GetMsg()` call, one of the ten A/ROSE primitives, is in this case an A/ROSE Prep service. `GetMsg()` returns a pointer to this message structure, which has already been partially initialized: `mId` is a statistically unique identification number for the particular message, and `mFrom` has already been filled in with the sender's task ID (the number returned by `OpenQueue()`, or by the utility `GetTID()`).

We need to identify the addressee in the `mTo` field and we need to specify `mCode = ICC_GETCARDS` (this constant is defined in the include file managers.h) in order to request information about Macintosh Coprocessor Platform cards in the machine. On receiving a message with this `mCode`, the ICCM expects in `mDataPtr` a pointer to 64 bytes (according to `mDataSize`), and fills the array cards[0..15] of tid_type for each slot with a value

- <0, if there is no Macintosh Coprocessor Platform card at all, or no ICCM
- =0, if there is an ICCM but no Namer Manager
- >0, if there is an ICCM and a Name Manager; the value is the Name Manager's TID

The rest is easy: For each Name Manager TID, a repeated call to `Lookup_Task()` returns successively all identifiers of tasks that registered correctly with the Name Manager. The variable `index` is initially set to zero and then passed by address; it is an internal value that must be passed back to A/ROSE unchanged on subsequent calls to `Lookup_Task()`. This call is an example of an A/ROSE utility call, which hides the underlying mechanism of sending a message with a specific `mCode` and `mDataPtr` to a manager, and getting the result back through a `Receive()` call.

Sending a message now to the Name Manager in the current slot with `mCode = NM_LOOKUP_NAME` and with `mDataPtr` pointing to an appropriate buffer, brings back the object name and type name of the task, which is finally displayed.

BUILDING A DOWNLOAD FILE

To download code to a NuBus card, you have to build a code resource. I will reproduce and discuss the required code (file osmain.c) in a simplified form here.

```
main ()
{
      struct ST_PB stpb, *pb; // Start parameter block.

      // Init OS with cMaxMsg messages and cStackOS stack.
      osinit (cMaxMsg, cOSStack);

      pb = &stpb;

      StartNameServer(pb);          // The Name Manager.
      StartICCManager(pb);          // You guess it!
      StartmyTask(pb);              // And our sample task.
      // Start all other required managers and tasks.

      // Start operating system.
      osstart (TICK_MIN_MAJ, TICKS_PS);
      // Should never get here!
} // Main().

void StartmyTask(struct ST_PB *pb)
{
      pb->CodeSegment = 0;
      pb->DataSegment = 0;
      pb->StartParmSegment = 0;
      pb->InitRegs.A_Registers [5] = GetgCommon() -> gInitA5;
      pb->ParentTID = GetTID();
      pb->stack = 4096;
      pb->heap = 0;
      pb->priority = 10;
      pb->InitRegs.PC = myTask;     // Entry point of myTask.
      if (StartTask (pb) == 0)// If the task does not get started,
            illegal ();             // go debugging.
}
```

The routines `StartNameServer(pb)` and `StartICCManager(pb)` are quite similar (except for slight variations in some parameters and the priority level) to `StartmyTask(pb)`.

So this is the code that will be downloaded to the card. The calls `osinit()` and `osstart()` are only meaningful in this context of an initial load of the card. The first call takes two parameters whose default values are `cMaxMsg = 500` (maximum number of available message buffers) and `cOSStack = 4096` (size of OS stack). In many cases, the `cMaxMsg` value in particular can be safely diminished, which allows optimization of memory usage on the card. The second call, `osstart (TICK_MIN_MAJ, TICKS_PS)` launches A/ROSE, with default values for the number of time-slicing ticks per second, and for a subdivision of major ticks into minor ticks.

In between these two calls, all other required tasks need to be initialized by means of their start parameter block; the required minimum consists of the Name Manager task (the linker finds its code under the name `name_server` in the library OS.o), and the InterCard Communication Manager task (again, its code is in OS.o). In our example, we added our own `myTask`, whose source code file is compiled separately (compare this with the routine `TaskProcessing()` in the TaskSample program):

```
staticchar  my_object_name [] = "myTaskName";
staticchar  my_type_name []  = "myTaskType";

void myTask()
// All it does at this point is to register with the Name Manager
// (in order to be recognized by possible clients looking for it)
// and then just send back the messages it receives.
{
      mMessage *m;

      if (!Register_Task (my_object_name, my_type_name, Machine_Visible))
            illegal (); // Go debugging: something mysterious happened.

      while(1)    // Forever !
      {
      m = Receive(OS_MATCH_ALL, OS_MATCH_ALL, OS_MATCH_ALL, OS_NO_TIMEOUT);
      if (m) {
          if (m->mStatus != 0) {   // What happened? A real program would
             FreeMsg(m);     // investigate but we'll just get rid of it
      }                      // here.
```

```
            else {
                  switch (m->mCode) {
                  case DUMMYCODE:
                        // Is there something to do with this message?
                        break;

      // Handle here all the message codes you specified in your design.

                  default:
                        m->mCode |= 0x8000;     // Unrecognized message code;
                        m->mStatus = OS_UNKNOWN_MESSAGE;
                                                // defined in "managers.h."
                        break;
                  } // Switch.

                  // Send message back.
                  SwapTID(m); // Swap mFrom and mTo fields.
                  m->mCode++; // Response is one greater, by convention.
                  Send (m);
                  }     // Message status was OK.
            }     // There was a message.
      }     // While.
}     // End myTask().
```

Finally, we need to put everything together. The following MPW shell commands do the trick. Adopting the convention on the A/ROSE distribution disks, we'll use the filename Start for what we will download. I recommend defining the MPW shell variables `AROSE`, `AROSEBin`, and `AROSEIncl` in a UserStartup file, which holds the corresponding folder pathnames.

```
C osmain.c -i "{AROSEIncl}"
C myTask.c -i "{AROSEIncl}"
Link -t 'DMRP' -c 'RWM ' ¶
      -o start ¶
      osmain.c.o ¶
      myTask.c.o ¶
      "{AROSEBin}"OS.o ¶
      "{AROSEBin}"osglue.o
```

Finally, we need to download the file to the available Macintosh Coprocessor Platform cards in your machine, by means of the Download tool:

```
"{AROSE}Downloader:Download"  start
```

The tool should reply with

```
Segment of size 00000040 is downloaded
Segment of size 00000054 is downloaded
Segment of size 00004D44 is downloaded
```

Now it's time to come back to our tool ShowTasks:

```
showtasks

slot = $0 :
$00000003: name "echo manager", type "echo manager"

slot = $D :
$0D000001: name "name manager", type "name manager"
$0D000003: name "myTaskName", type "myTaskType"
```

. . . and to go ahead and send messages to myTask on a Macintosh Coprocessor Platform card. To try this, launch ClientAppli again, but this time without the TaskSample application running. ClientAppli now finds the "server" named `myTaskName` in its slot, and messages sent to it are returned as expected.

A MANDELBROT SETS EXERCISE

The downloaded file in the Macintosh Coprocessor Platform card works, but it gets boring fast: our server task is quite lazy, and doesn't do anything besides echoing our messages. For a more interesting programming exercise, open the MCPMB folder. The program you'll find there computes Mandelbrot (MB) sets in parallel processing, involving as many Macintosh Coprocessor Platform cards as you can put into your machine.

For each line to be computed, a message is sent to an MBTask, carrying along the required parameters and a pointer to where the line fits into the bitmap. The MBTask allocates a local buffer each time (for pedagogical reasons, I didn't optimize the design too much) and sends the computed data back over the NuBus by means of a NetCopy() call. Have a look at the source code on the CD, play with it, and let me know what you did to improve on the error handling and some other flaws in it.

THAT'S ALL, FOLKS . . .

This article has given you an idea of how to find your way around the Macintosh Coprocessor Platform and A/ROSE. You now know something about the origins, architecture, and implementation of this generic hardware and software foundation, and have seen some samples of A/ROSE programming. If you want to go on from here, the *APDAlog*® contains everything you need to know to order the complete Macintosh Coprocessor Platform Developer's Kit, or the A/ROSE Software Kit, or just the *Macintosh Coprocessor Platform Developer's Guide*.

Thanks to Our Technical Reviewers
Ned Buratovich, Dave Comstock, Chris Deighan, Ken Siemers, Steve Wang.
A special thanks to Anumele D. Raja. •

THE PERILS OF POSTSCRIPT— THE SEQUEL

Developers are discovering the advantages of using PostScript® dictionaries in applications, but along with the advantages come some perils. One peril awaits if you download a dictionary using PostScriptHandle. Another can trip you up after downloading a dictionary if you then download a font using the SetFont procedure I described in develop, *Issue 1. How to avoid these perils? Read on to learn some tricks for dicts in picts.*

SCOTT "ZZ" ZIMMERMAN

More and more developers are beginning to use direct PostScript code in their applications. In my "Perils of PostScript" article in develop, Issue 1, I addressed a couple of problems that arise when you use PostScript code to print documents. In this sequel, we'll look at some problems you will encounter if you attempt to use PostScript dictionaries in your applications.

ABOUT POSTSCRIPT DICTIONARIES

A PostScript dictionary is a collection of predefined variables and/or procedures. Using a PostScript dictionary can significantly reduce the size of the PostScript code generated by your application and make it more efficient. For instance, consider a large PostScript file in which the operator `currentpoint` is used frequently. You can define in your dictionary a PostScript procedure called `cp` that makes a call to the `currentpoint` operator. You can then replace `currentpoint` with `cp` throughout the file, thus reducing its size. Similarly, by defining a PostScript procedure to represent a series of operators, you can express a compound operation much more efficiently. And storing procedures in a dictionary that you create can also prevent you from inadvertently redefining something that has already been defined.

One great example of a PostScript dictionary is the one used by the LaserWriter® driver, variously called LaserPrep (after the file it resides in, at least until System 7), AppleDict (the Apple name for it), and good ol' md (the PostScript name for it, and the one I prefer to use). The LaserWriter driver generally uses one or more md

SCOTT "ZZ" ZIMMERMAN is a DTS printing guru. After two and a half years at Apple he says he's particularly impressed with the strictly enforced dress code. In his spare time he sails, scuba dives for lobsters, and plays the piano, guitar, and saxophone. Zz has a penchant for pets. His doorway is adorned by a melted gummy rat, a good luck charm left over from his Intel days. At home, atop his monitor is perched a rare Asian black scorpion (behind glass, we hope). His other cuddly pets include two geckos and a lovable iguana. •

routines to perform a particular QuickDraw operation. (See the sidebar on the next page for a review of how the LaserWriter driver works.) For example, a call to the QuickDraw `CopyBits` routine is translated by the driver into a call to the `db` or `cdb` operators stored in md. As another example, during font downloading the LaserWriter driver uses `bn` and `bu`, both stored in md, to call `save` and `restore` (`bn` calls save and `bu` calls restore).

If you want to record a piece of PostScript code that references procedures contained in a dictionary, you must also record the dictionary. I describe how to download a dictionary, and how to avoid the pitfalls involved, in the next section.

Once your dictionary has been downloaded, you should be able to continue to reference it until the end of the job. But alas, this is not so, at least until the new printing architecture ships sometime after System 7. Under the current architecture, font downloading interferes with PostScript dictionaries. I discuss this problem and how to get around it under "The Perils of Font Downloading."

THE PERILS OF DICTIONARY DOWNLOADING

One of the easiest methods for downloading a PostScript dictionary is by using the `PostScriptHandle` picture comment. You can use this comment to download directly to the LaserWriter a block of PostScript code stored in a handle. (See Technical Note #91, Optimizing for the LaserWriter—PicComments, for more information.) When you use the `PostScriptHandle` comment, you must insert the `PostScriptBegin` and `PostScriptEnd` picture comments around the block of PostScript code you are trying to download, like this:

```
PicComment(PostScriptBegin, 0, NIL);
      (************************************************************)
      (*** Your PostScript representation of document goes here.***)
      (************************************************************)
PicComment(PostScriptHandle, size, handle);
      (***********************************************************)
      (*** Your QuickDraw representation of document goes here.***)
      (***********************************************************)
PicComment(PostScriptEnd, 0, NIL);
```

As described in my first article, the `PostScriptBegin/End` comments are markers that ensure that the right piece of code will execute on the right device. When the LaserWriter driver sees `PostScriptBegin`, it ignores all QuickDraw drawing calls and just executes picture comments. When a `PostScriptEnd` is received, the LaserWriter driver will once again interpret QuickDraw calls. So when printing to a LaserWriter printer, only the picture comments are executed, while the QuickDraw code between `PostScriptBegin` and `PostScriptEnd` is ignored.

The definitive references on the PostScript language are the *PostScript Language Reference Manual* (Addison-Wesley, 1985, available from APDA—#T0182LL/A), the *PostScript Language Tutorial and Cookbook* (Addison-Wesley, 1985), and *PostScript Language Program Design* (Addison-Wesley, 1988). •

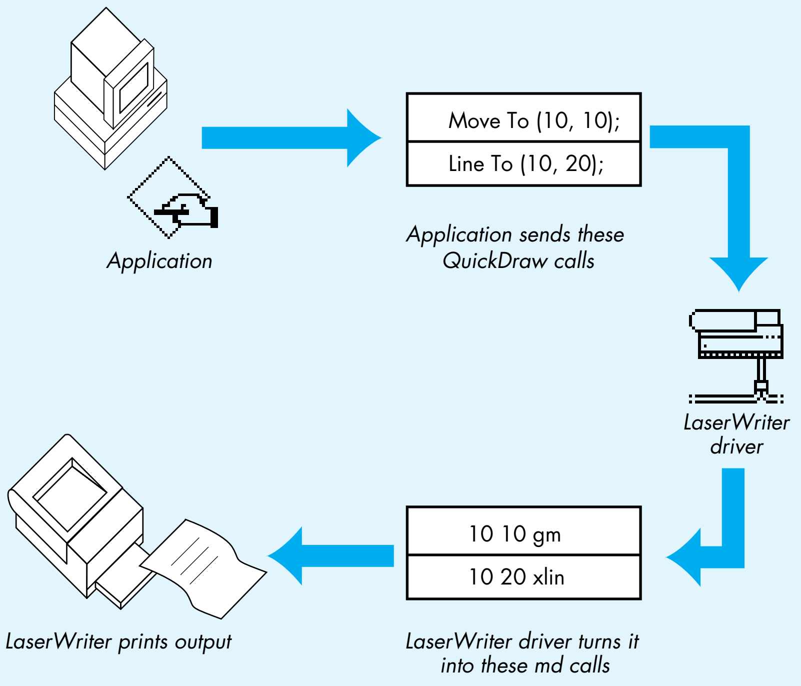
A REVIEW OF HOW THE LASERWRITER DRIVER WORKS

The LaserWriter driver is a complex piece of software that handles communications between an application and the LaserWriter printer. To print a document, the application opens the Printing Manager, which in turn loads and initializes the LaserWriter driver. The application then makes standard QuickDraw calls similar to those used to render the document on the screen. The LaserWriter driver intercepts these calls and converts them into the equivalent PostScript code for rendering the document on the LaserWriter printer. (See Figure 1.)

In some cases, one QuickDraw operation translates into one PostScript operation, but more frequently, the QuickDraw operation translates into several PostScript operations. To abbreviate these operations, the LaserWriter driver stores them as procedures in a PostScript dictionary.

When the LaserWriter driver first connects to the LaserWriter printer, it checks to see if its dictionary exists and if the version of the dictionary matches the version of the driver being used. If not, it downloads the correct dictionary before proceeding. (This is what the message "initializing printer" means when you print for the first time after turning on the printer.)

Once the correct dictionary is in place, the job of translation becomes much easier. Each QuickDraw operation now becomes one line of PostScript code, referencing a procedure defined by the dictionary. Complex QuickDraw operations (like font downloading) still require many lines of PostScript code, but in general, the translation is one for one. Since the QuickDraw code is translated rather than rendered, the LaserWriter driver doesn't need to spool the data to disk. Instead, each operation is translated and sent to the printer as it is received.



Application
Move To (10, 10);
Line To (10, 20);
Application sends these QuickDraw calls
LaserWriter driver
10 10 gm
10 20 xlin
LaserWriter driver turns it into these md calls
LaserWriter prints output

Figure 1
How the LaserWriter Driver Works

But `PostScriptBegin` and `PostScriptEnd` also save and restore at least part of the state of the device, which can cause problems for your dictionary. To avoid this, you should use the picture comment `PostScriptBeginNoSave` (comment kind = 196) to prevent the save and restore from occurring, like this:

```
      (**************************************************)
      (*** Your definition of the dictionary goes here.***)
      (**************************************************)

PicComment(PostScriptBeginNoSave, 0, NIL);
      PicComment(PostScriptHandle, dictsize, dicthandle);
PicComment(PostScriptEnd, 0, NIL);
      (*********************************)
      (*** Now you send the document. ***)
      (*********************************)

PicComment(PostScriptBegin, 0 NIL);
      (***********************************************************)
      (*** Your PostScript representation of document goes here.***)
      (***********************************************************)
PicComment(PostScriptHandle, size, handle);
      (**********************************************************)
      (*** Your QuickDraw representation of document goes here.***)
      (**********************************************************)
PicComment(PostScriptEnd, 0, NIL);
```

If you don't need to export your dictionary into picture files, you can get the LaserWriter driver to auto-download your dictionary by keeping the dictionary code in a PREC(103) resource. After the LaserWriter has saved its state, it does a blind `GetResource` (that is, from any open resource file) on PREC(103). If one is found, it is downloaded to the printer after the md dictionary, and before the job. You can use this method of downloading for dictionaries that are used only to contain state information about the current job. (When a graphic is copied onto the clipboard, only the state information, not the entire dictionary, is required for the code to execute.)

However, use of the PREC(103) resource does have some limitations. It only works at print time, and there can be only one. That is, the LaserWriter driver does *not* attempt to download all the PREC(103) resources in all the open resource files. The first one it finds wins. (This method of downloading dictionaries is documented in Technical Note #192, Surprises in LaserWriter 5.2 and Newer.)

THE PERILS OF FONT DOWNLOADING

In my "Perils of PostScript" article in Issue 1 of develop, I showed a small procedure called `SetFont` that downloaded a font using QuickDraw, while maintaining the ability to reference that font using PostScript. The problem with that method is that the process of font downloading executes the PostScript `restore` operator. This operator restores the state of the printer to a state that was saved before your dictionary was defined. Because of this, any reference to your dictionary is lost.

Another way to understand what happens in this case is to look at what the LaserWriter driver does during printing. At the start of a print job, the LaserWriter driver configures the LaserWriter's graphics state to look more like QuickDraw. This includes moving the origin (0,0) from the bottom left (PostScript style) to the top left (QuickDraw style), and setting the default resolution to 72 dpi. After the driver has configured the printer, it performs a save, which saves the complete state of the device. The driver then begins downloading the rest of the job, containing the PostScript code generated by the LaserWriter as well as any additional PostScript code sent by the application.

The LaserWriter driver fully restores the state of the device, by executing the PostScript `restore` operator, before downloading a font. During font downloading, the characters of the font are actually defined, sometimes using normal PostScript drawing operators. Because of this, the LaserWriter driver restores the state of the printer before defining the characters. Once the characters have been defined, the state is saved again. This way, the LaserWriter driver can assume it knows the state of the device. Since the state saved by the LaserWriter driver does not contain any of the symbols defined by the application, all of them are lost after any attempt to download a font.

WHICH WAY OUT?

Now that we understand the problem, let's discuss potential solutions. The `restore` operator affects everything that has changed except two areas: some of the PostScript stacks (specifically the operand, dict, and execution stacks), and the contents of PostScript strings. This suggests that to save small units of information, you can simply push them onto the stack, or convert them and store them as PostScript strings.

Unfortunately, it's not quite that easy.

PostScript makes a distinction between simple and composite objects. Simple objects (like numerical values and booleans) contain their value within the object. Composite objects (like strings, procedures, and dictionaries) contain only a pointer to the real data, which is stored elsewhere in PostScript Virtual Memory. Simple objects on the stack are indeed preserved across a `restore`, but if there are composite objects on the stack that are new (that is, newer than the state being restored), an invalidrestore error is generated. If your dictionary only contains simple objects, then you can indeed push each of the variables defined in the dictionary onto the stack separately and rebuild the dictionary after the restore. The overhead here is obviously enormous, though, and most useful dictionaries contain procedures and/or strings, rendering this technique useless.

In the case of strings being preserved across a restore, let me quote from the *PostScript Language Reference Manual*, p. 44: "In the current PostScript design, restore actually does not undo changes made to the elements of strings. We consider this behavior to be a defect, and do not recommend that PostScript programs take advantage of it." Beyond this easily ignored admonishment, though, is another problem. The strings in question must be preexisting: strings you create just before the restore will, of course, be destroyed by the restore, or, if they are on the stack, will cause an error. You could probably find some scratch strings in one of the standard dictionaries to use, but this is not recommended, for obvious reasons.

AN END TO BN AND BU

Another way to solve the problem would be to redefine save and restore to not do anything. This way, font downloading would not cause the state to be restored. This would make the application developer responsible for preserving the state, which is easily done using other PostScript operators. But unfortunately, the definitions of save and restore cannot be changed without exiting the server loop. That is, you cannot override their definitions from within a job. Because of this, you have to fall back on plan B: override the operators that call save and restore. In the case of font downloading, these operators are bn and bu, as mentioned earlier.

This method is the most widely used solution to our problem, has the fewest limitations, and is the method recommended here. Please note, however, that tinkering with md operators outside of this specific use is strongly discouraged. (See the sidebar on the next page.)

The main job of bu and bn is to preserve the state of the PostScript device. As long as your PostScript code preserves the state, these calls aren't even required. In the fragment that follows, we first create our own dictionary, called mydict, with room for ten symbols, although we don't define them all. Next we define killbu. killbu is responsible for first saving the old definition of the bu routine, and then setting its value to the empty procedure ({}), which does nothing. The original definition of bu is simply pushed onto the stack. Next we write a routine restorebu, to restore the definition of bu when we are through. This routine is responsible for popping the original value off the stack and storing it back into the bu symbol; it assumes that the definition of bu is on the top of the stack. Then we define two similar routines, killbn and restorebn, which take care of the bn

WARNING: CALLING MD ROUTINES MAY BE HAZARDOUS TO YOUR CODE

Many developers have started to call md routines from within the PostScript code generated by their applications. This is dangerous, for a number of reasons.

The first is that the md dictionary is defined and maintained by the LaserWriter driver. This means that it is always subject to change, and code that depends on the md dictionary must be version dependent. This is possible, but far from elegant.

Another problem with using md operators is that they may not work the same way on all devices. Remember that the LaserWriter driver is used to drive a lot more devices than just an Apple LaserWriter.

Use of md operators has already led to compatibility problems with major applications, and most developers have realized the danger in using them. The easiest way to avoid problems with these routines is to not call them.

If you really need the functionality of a particular md operator, simply redefine it in your own dictionary. Using tools like LaserTalk (formerly from Emerald City Software, now from Adobe), you can "disassemble" md operators back to their PostScript primitives. You can then redefine them using a different name in your own dictionary. Now you have a routine that does exactly what the md routine did, but you remain in control of its definition. Most of the md operators are very small, so the storage penalty of redefining them in your own dictionary is minimal.

Now that I've warned you, I'm going to show you how to tinker with two operators stored in md: bn and bu. All routines, including these two, are subject to change; by special arrangement with engineering, bn and bu will change in a compatible way, but this is *not* true for any of the other routines defined in md. This article shows a specific use of bn and bu, and checks for their existence before attempting to access them. This is *not* meant to endorse other uses of these or any other md routines.

operator. Finally, we define a fun little routine to call to make sure our dictionary is actually being preserved after font downloading. We call this one `titleshow`. So now we have a dictionary, all ready to use.

```
SendPostScript('/mydict 10 dict def');
SendPostScript('mydict begin');
SendPostScript('/killbu {//md /bu get //md /bu {} put} def');
SendPostScript('/restorebu {//md exch /bu exch put} def');
SendPostScript('/killbn {//md /bn get //md /bn {} put} def');
SendPostScript('/restorebn {//md exch /bn exch put} def');
SendPostScript('/titleshow {dup gsave');
SendPostScript('currentscreen 3 -1 roll pop 120 3 1 roll setscreen');
SendPostScript('.5 setgray show grestore true charpath gsave');
SendPostScript('1 setlinewidth 0 setgray stroke grestore');
SendPostScript('.5 setlinewidth 1 setgray stroke }def');
SendPostScript('end');
```

Okay, now that we have the routines for killing `bu` and `bn`, we need to call them. It's very important at this point to check for their existence before attempting to alter their definitions. This is because, as mentioned earlier, the new printing architecture that will ship sometime after System 7 will handle font downloading differently. The `bu` and `bn` operators will no longer exist; in fact, it's not clear that the md dictionary will still exist. The following PostScript commands check for the existence of both the dictionary and the symbol. If they don't exist, our code assumes it is running under the new printing architecture, and does nothing to insulate the dictionary. The code fragment executes fine on LaserWriter drivers up to and including System 7.0. It has also been tested in both foreground and background. Considering the future of `bn` and `bu`, it is very likely that this code will continue to work even under the new printing architecture. Here, then, is the code to check for and kill `bn` and `bu`:

```
SendPostScript('mydict begin');
SendPostScript('//md /bu known {killbu} if');
SendPostScript('//md /bn known {killbn} if');
SendPostScript('end');
```

Pretty straightforward: if the routine exists, call the correct routine to kill it. The most important thing to note here is the order of the routines. Since `killbu` and `killbn` push things onto the stack, `restorebu` and `restorebn` must be called in opposite order to get the correct results. So after the job is finished, we call:

```
SendPostScript('mydict begin');
SendPostScript('//md /bn known {restorebn} if');
SendPostScript('//md /bu known {restorebu} if');
SendPostScript('end');
```

TO SUM IT ALL UP

PostScript dictionaries are useful because they can significantly reduce the size of the PostScript code generated by your application, and can be exported into pictures. Perhaps the easiest way to record PostScript into a picture is by using the `PostScriptHandle` picture comment. In this case, remember to use the `PostScript BeginNoSave` comment to prevent `PostScriptBegin` and `PostScriptEnd` from saving and restoring at least part of the state of the device, which can cause problems for your dictionary. To prevent font downloading from interfering with your PostScript dictionaries, you can override `bn` and `bu`, the PostScript operators that call `save` and `restore`. Outside of this solution, you should absolutely avoid using md operators.

The code included in the Perils of PS II folder on the *Developer Essentials* disc is basically the same code that has been shown here, rolled into an application shell that opens and initializes the Printing Manager. Also included is the definition of the `SendPostScript` procedure referenced in this article.

Thanks to Our Technical Reviewers
Pete "Luke" Alexander, Jay Patel, David Williams •

DRIVING TO PRINT: AN APPLE IIGS PRINTER DRIVER

Do you have a printer that would print awesome text and graphics if only someone would write a driver for it? Have you looked at the driver specifications and become hopelessly confused? If you want to give your Apple IIGS some expanded printing capabilities, don't put this issue down until you've read this article!

MATT DEATHERAGE

In theory, printer drivers seem like a great solution. All you have to do is drop a printer driver file in your Drivers folder, and all of a sudden you'll be able to create dazzling text and graphics from whatever desktop application and on whatever kind of printer you happen to use with your Apple IIGS. No more writing to printer manufacturers or waiting for application upgrades to support your printer.

Unfortunately, the reality isn't quite as nifty as the theory. Even though Apple released printer driver specifications in early 1988 (just before System Disk 3.2), only a few third-party printer drivers have surfaced. The specifications are complex and sometimes confusing, and they have not always been accurate. Most of all, printer drivers are intrinsically complicated and difficult to develop. The driver has to do all of the work in getting images printed, with no imaging help from the Print Manager.

This article explains the mysteries of the printer driver: what it does, how it does it, and how to write one. To illustrate the concepts, we've provided a sample printer driver called Picter. Picter takes the image to be printed and saves it to your boot disk as a QuickDraw II picture file. Picter allows you to literally print a graphic document to disk. Much of Picter's structure and code is directly applicable to any printer driver. What's more, the dialog routines in Picter, which are very similar to those in the new ImageWriter and ImageWriter LQ drivers released with System Software 5.0.3, will enable you to be consistent and stylish in your user interface. You will find Picter in the IIgs Printer Driver folder on the *Developer Essentials* disc.

MATT DEATHERAGE used to think he was a cynic, but two and a half years in Developer Technical Support for the Apple II has made him doubt even that. His perpetual quest for sleep has been interrupted by his new responsibility for the ProDOS partition on the Developer CD and *Developer Essentials* disc, as well as by his resuming the role of DTS technical lead for Apple IIGS system software. It would be enough to make his head spin, he says, "if my head were jointed that way." His musical pursuits continue with work on an album, *The Fruited Computer Follies of 1990,* which will never be released as all of the songs on it are entirely unsuitable for polite company. He's currently conspiring with Robert Thurman to withhold the definition of "PPG." •

HOW PRINTING WORKS

Printing from a desktop application appears to be a black box. You make some Print Manager calls and *voila!*—there's a piece of paper with a printed image of what you drew. The Print Manager uses some serious magic to turn your image into ink on paper, but that's all hidden from the application.

So now that we know what we're getting into, let's briefly review how applications print through the Print Manager.

WHAT THE APPLICATION SEES

In the Apple IIGS desktop environment, documents are kept in windows, which are extended versions of the QuickDraw II drawing environment—the `Grafport`. The features defined by the `Grafport` include where drawing will and will not occur, what size pen will be used to draw lines and other objects, what method will be used to draw them, what colors and patterns will be used with the objects drawn, what style, size, font, and colors will be used for text drawing, and where the image resides in memory.

The model for printing is quite similar to drawing in a window. Instead of drawing into a window `Grafport`, your application draws into a *printing* `Grafport`, which defines the drawing environment for a single page. The clipping and visible regions (the `clipRgn` and `visRgn`) are set to the rectangular area of the page, for example.

An application prints by drawing into a printing `Grafport`, which it obtains by opening a printing document with the Print Manager call `PrOpenDoc`. The Print Manager responds by returning a printing `Grafport` in which the material to be printed should be drawn. The printing `Grafport` is initialized at the beginning of each new page (signified by `PrOpenPage`). The application then draws the page, closes it (with `PrClosePage`), and repeats this sequence until all pages have been printed. The application then closes the document (with `PrCloseDoc`) and prints any images the driver may have spooled with `PrPicFile`. The sequence of calls starting with `PrOpenDoc` and ending with `PrPicFile` is referred to as the *print loop*, since the middle calls (`PrOpenPage` and `PrClosePage`) are repeated once for each page to be printed. Note that `PrPicFile` should *always* end the print loop.

HOW IT REALLY WORKS

If the Print Manager does all this for the application, as the *Apple IIGS Toolbox Reference* says it does, where does a printer driver fit in?

To understand how printer drivers work, you first need to realize that the preceding description of how applications print is exaggerated. Everything listed above as done by the Print Manager is really done by the currently selected printer driver. Although the calls are Print Manager calls, the only action the Print Manager takes on these calls is to make sure the printer driver is available and to dispatch the calls to the driver. The application model says this work is done by the Print Manager to prevent application dependency on any particular driver. From the application's point of view, the Print Manager's role in printing allows the application to be independent of any particular driver. But in reality, your printer driver will handle all the work associated with several of these "Print Manager" calls.

While at first it might seem like a cop-out by Apple to require the printer driver to handle all the work in the print loop, this strategy actually makes a lot of sense. The printer driver must ultimately transform images into ink on paper, so for maximum flexibility Apple has given the printer driver control over the entire imaging process, from the opening of a document to the printing of spooled images. Since no one but the printer driver author knows what user-selectable features the driver will support, the printer driver should be responsible for the style and job dialog boxes through which these features will be chosen. And because the printer driver knows how to best handle internal errors, it's a good idea to make it responsible for returning and accepting error codes from the application.

Although the printer driver has to handle all the imaging, the Print Manager does provide a lot of support for other parts of the printing process. One of the tasks the Print Manager supports is communication—once an image has been converted into printer codes, the codes have to be sent to the printer. The Print Manager keeps track of a different kind of driver—the *port driver*—that handles this communication with the printer through the internal ports of the Apple IIGS (or through the slot-based peripherals). The port driver essentially relieves the printer driver of the work of communicating with the printer. All the printer driver has to do is ask the port driver to read or write data to the printer, and the port driver handles all the details. The Print Manager also keeps track of which printer and port drivers the user has chosen with the Control Panel desk accessory.

Figure 1 shows the relationship of the printer driver and the port driver to the Print Manager. The Print Manager handles some duties alone while passing others directly through to the printer or port driver.

THE PRINT RECORD

Since the printer driver does all the interesting imaging work, it has to have some way to exchange vital information with the application. Applications need to know the size of the pages to be printed so that they can paginate properly. They may need to know the vertical sizing factors so that better resolution graphics can be printed when higher resolutions are available. Or they may need to know the resolution of the printer for precise printing chores. This information is communicated through a data structure known as the *print record*. The print record is associated with every document to be printed, and it is the only way the printer driver can keep these parameters associated with a document.

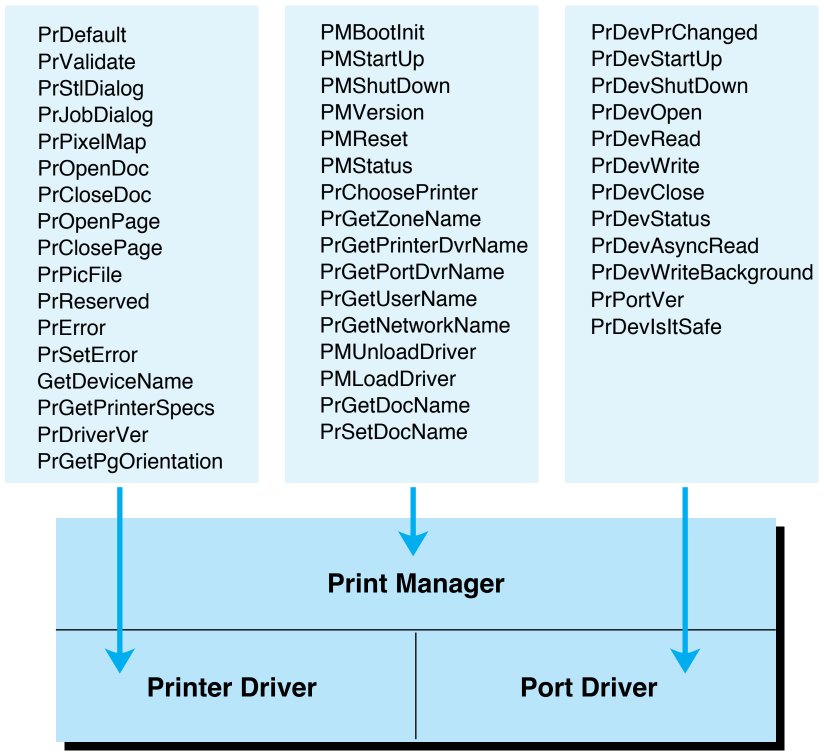
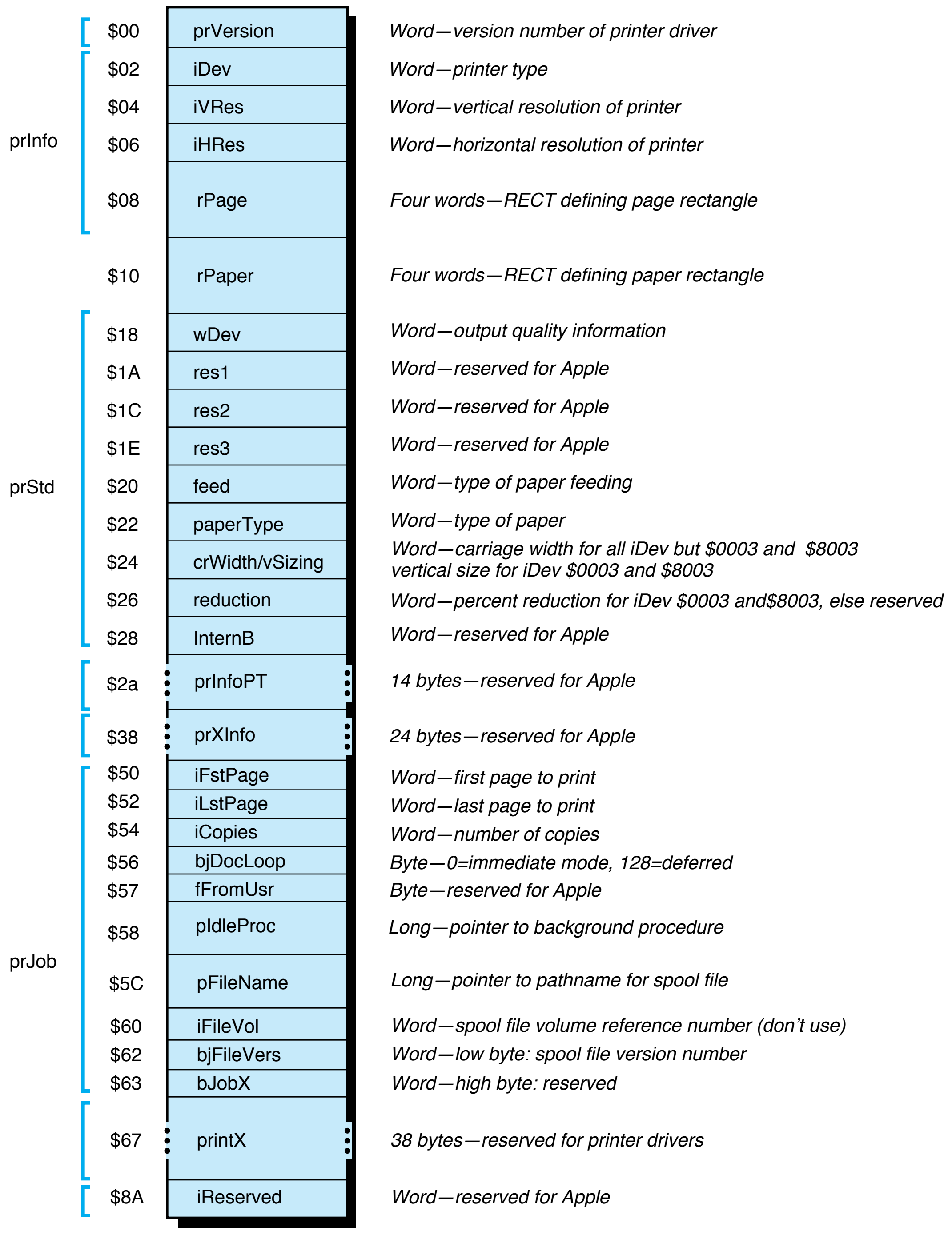


Figure 1
Print Manager Calls

Figure 2, on the next page, shows the print record in fully documented detail. Notice that some fields are marked simply as reserved—that means reserved for Apple. Using these fields is a really good way to make your application not print with other drivers or to make your driver not work with future system software.

The print record contains all the parameters associated with a printing job. It includes not only the page and paper sizes and the resolution of the printer and other hardware parameters, but also the values selected by the user in the Page Setup and Print dialog boxes, which are presented by the printer driver. The print record contains all the information necessary to print a document the same way as many times as necessary.



prInfo	$00	prVersion	*Word—version number of printer driver*
prInfo	$02	iDev	*Word—printer type*
prInfo	$04	iVRes	*Word—vertical resolution of printer*
prInfo	$06	iHRes	*Word—horizontal resolution of printer*
prInfo	$08	rPage	*Four words—RECT defining page rectangle*
	$10	rPaper	*Four words—RECT defining paper rectangle*
prStd	$18	wDev	*Word—output quality information*
prStd	$1A	res1	*Word—reserved for Apple*
prStd	$1C	res2	*Word—reserved for Apple*
prStd	$1E	res3	*Word—reserved for Apple*
prStd	$20	feed	*Word—type of paper feeding*
prStd	$22	paperType	*Word—type of paper*
prStd	$24	crWidth/vSizing	*Word—carriage width for all iDev but $0003 and $8003 vertical size for iDev $0003 and $8003*
prStd	$26	reduction	*Word—percent reduction for iDev $0003 and$8003, else reserved*
prStd	$28	InternB	*Word—reserved for Apple*
	$2a	prInfoPT	*14 bytes—reserved for Apple*
	$38	prXInfo	*24 bytes—reserved for Apple*
prJob	$50	iFstPage	*Word—first page to print*
prJob	$52	iLstPage	*Word—last page to print*
prJob	$54	iCopies	*Word—number of copies*
prJob	$56	bjDocLoop	*Byte—0=immediate mode, 128=deferred*
prJob	$57	fFromUsr	*Byte—reserved for Apple*
prJob	$58	pIdleProc	*Long—pointer to background procedure*
prJob	$5C	pFileName	*Long—pointer to pathname for spool file*
prJob	$60	iFileVol	*Word—spool file volume reference number (don't use)*
prJob	$62	bjFileVers	*Word—low byte: spool file version number*
prJob	$63	bJobX	*Word—high byte: reserved*
	$67	printX	*38 bytes—reserved for printer drivers*
	$8A	iReserved	*Word—reserved for Apple*

Figure 2
The Expanded Print Record

PRINTING MODES

In addition to being concerned about what to print, you must be concerned about the way in which it's printed. There are two modes for printing. The differences between these modes amount to two different models for printing.

Immediate mode. When you print in *immediate mode,* every page is printed as it's defined. The driver does not store an image of the page before it sends it to the printer. This strategy can limit the driver's options when printing a page. To see how, you first have to understand how immediate mode works.

When QuickDraw performs a graphic operation, it calls a standard set of low-level routines to do it—the QuickDraw bottleneck procedures. A pointer to them exists in every `GrafPort`'s `grafProcs` field, where a value of 0 means that QuickDraw should use the standard procedures. This is briefly mentioned in Technical Note #35, but it is covered in great detail in the note just preceding it: Apple IIGS Technical Note #34, Low-Level QuickDraw II Routines.

To print in immediate mode, you install your own set of bottleneck procedures into the printing `GrafPort.` When the application draws any object into the printing `GrafPort`, QuickDraw calls your bottleneck routines to actually image that object.

Because immediate mode printing responds to object-drawing commands sent by QuickDraw, immediate mode printing works best for target devices that handle similar objects. For example, the LaserWriter has built-in PostScript code that can image objects in much the same way QuickDraw does. The LaserWriter driver installs bottleneck procedures that convert QuickDraw objects into PostScript objects and sends them immediately to the LaserWriter, printing the page when the page is closed with `PrClosePage`.

Unfortunately, most printers do not handle graphic objects. The graphics capabilities of most printers are of the "print a dot of this color at this location" variety. To print images to these devices, a driver has to convert the images into printer codes that place the dots where they need to go. Doing this properly requires waiting until all objects are drawn on the page before sending any codes to the printer. If you try to image and print each QuickDraw object as it's drawn, you'll get the wrong results when the application draws white pixels on top of previously colored pixels. (You will also have to move the paper backward and forward enough to inspire demonic possession stories.)

Because of this limitation, many dot-matrix printers ignore graphic objects when printing in immediate mode, transforming only text drawing into simple ASCII text printing using the printer's built-in font. Since this is not what you see on the screen, immediate mode printing is often referred to as *draft mode*, even though immediate mode printing can be of excellent quality on the right target device.

SOME THINGS TO KNOW ABOUT PRINT RECORD FIELDS

Volume 1 of the *Apple IIGS Toolbox Reference* does a good job explaining most of the fields in the print record, but it contains some incomplete information. One such omission occurs in the *Reference's* description of the `iDev` field. The `iDev` field identifies the kind of printer. The *Reference* lists two values for this field—ImageWriter and LaserWriter—which leads to problem code in applications such as the following:

```
if PrintRecord.iDev = 1 then
     {It's an ImageWriter}
else
     {It's a LaserWriter}
endif
```

In reality, there are at least six defined values for `iDev`:

$8001	ImageWriter
$0002	ImageWriter LQ
$0003	LaserWriter
$0004	Epson
$8001	Generic dot matrix printer
$8003	Generic laser printer

The $8001 and $8003 `iDev` values are provided for generic compatibility. If a driver has an `iDev` of $8001, it interprets the style subrecord of the printer record as is documented for the ImageWriter driver. If the `iDev` is $8003, it interprets the style subrecord as it would for the LaserWriter driver.

Unfortunately, because this is the only device identification field present in the print record, there is no way to uniquely identify printers assigned to these values. For instance, suppose you have two printers with printer drivers in your system—the GlopJet and the ImageStamper. Both drivers use an `iDev` of $8001. Applications are encouraged to save print records with documents so that the user's print settings are maintained across sessions. If you open a document with a print record created by the GlopJet driver but your currently selected printer is the ImageStamper, the ImageStamper driver will be passed the print record and asked to validate it. The ImageStamper driver looks at `iDev` and sees $8001, and it has no other way to know that this print record is not an ImageStamper record.

Drivers with unique `iDev` values don't have this problem. For example, the LaserWriter driver knows that if the `iDev` value isn't $0003, it's not a LaserWriter print record and should be filled with default values.

Apple's Developer Technical Support group will assign new `iDev` values to printer driver authors if neither the $8001 or $8003 interpretation of the style subrecord is suitable, but you must be aware that some applications

Deferred printing. Since immediate mode printing is not suitable for graphics on many printers, most printing jobs will be deferred. In *deferred* or *spool mode*, everything that is drawn is captured to be printed later. Text is imaged together with graphics to return as accurate a reproduction of the document as possible.

How the printer driver captures the image is entirely discretionary. If you like, you can attach a pixel map large enough for the entire page to the printing `GrafPort` and let the application draw the page into the pixel map. This method would give you a premade pixel map, waiting for you to transform it into printer codes and send it out. At screen resolution, however, a full U.S. letter-sized page would take just will not work with other formats of style subrecords. It's better for compatibility purposes to support one of the existing style subrecord formats if possible.

For instance, some applications don't like to let the user choose items that don't work very well. If an application doesn't print very well without color, it might do something unfortunate like set the "color" bit in the `wDev` field before starting the printing loop. If the driver doesn't support color printing, it will catch this error in the `PrValidate` routine and may reinitialize the print record with default values. If the driver author is lucky, the application will first check the `iDev` field to make sure that the "color" bit is supported in the style subrecord. If you're really lucky, the application will call `PrGetPrinterSpecs` and keep out of the print record altogether. Many applications just blast the bit.

WHAT DRIVERS AND APPLICATIONS SHOULD DO

To keep your handling of the print record kosher, there are a few things you should keep in mind.

First of all, since print records are associated with printing jobs, it would be nice to keep all parameters that go with a printing job in the print record. But since a field is either defined or reserved, it's not clear where you can put a new parameter. If your printer has 14 different internal fonts and you want the user to choose one of them, where can you put that information?

Apple has set aside the 38 bytes in the print record labeled `printX` for printer driver use. Nonstandard parameters and values can go there. This area is left to the discretion of the printer driver. It will always remain a miscellaneous storage area, no matter what Apple does with it in drivers it develops, and its interpretation will not depend on the `iDev` field. In other words, if the LaserWriter driver stores a parameter there, drivers with $8003 `iDev` values are not expected to do the same.

Applications absolutely must not tamper with the `printX` subrecord nor try to interpret any items in there. Applications have most of memory for parameters, while printer drivers only get these 38 bytes in the print record. Applications, keep out.

It's also important that neither drivers nor applications alter the print record fields marked reserved for Apple— in particular the `prInfoPT` and `prXInfo` subrecords. Older versions of Apple's drivers stored a private copy of the `prInfo` subrecord in `prInfoPT` (PT stands for "private"). Discovering this fact, some applications used this copy instead of the original. Since this feature was never documented, however, relying on it is likely to make your application not work with other drivers. As for the `prXInfo` subrecord, it may be defined in the future for the storage of parameters between spooling and printing (between `PrCloseDoc` and `PrPicFile`).

over 56K of contiguous memory. That's per page—a 20-page document would require 20 such blocks.

For this reason, most printer drivers (including Picter and Apple's drivers) use QuickDraw pictures to capture the images. Pictures are an encoded history of the QuickDraw calls used to create an image. When you play back a picture using the QuickDraw auxiliary call `DrawPicture`, QuickDraw does all the drawing necessary to recreate the image. Instead of taking 32K to store a screen-sized rectangle filled with a given pattern, a picture stores the same information in the few bytes that encode the pattern, the rectangle size, and the "paint" command.

Because pictures contain recorded QuickDraw II objects, they can be redrawn at different resolutions or in different proportions with excellent results. If you call `DrawPicture` with a destination rectangle of a different size than the one the picture was recorded with, QuickDraw's picture algorithms are capable of changing the sizes and proportions of every object in a picture to match the changed destination rectangle.

This intelligent scaling behavior makes pictures perfect for the needs of most printer drivers. Since most printers are capable of screen resolution that is better than that of the Apple IIGS (80 pixels per inch horizontally by 36 pixels per inch in 640 mode), some kind of scaling will be necessary to create screen resolution images at the proper size regardless of resolution changes. For example, to achieve an image of the proper size when your target device supports 160 dpi horizontally by 72 dpi vertically, you'll need two printer pixels in each direction to represent one screen pixel.

Simply magnifying each screen pixel to be the appropriate number of printer pixels gives the image the right size, but the resolution is still the same as the screen's. To get *better* resolution, QuickDraw's picture algorithms are a good choice. For our sample target device that supports 160 dpi horizontally by 72 dpi vertically, your driver could call `DrawPicture` to image the stored page-picture in a rectangle twice as large as was used to record the picture. QuickDraw will then draw all the objects in the picture at twice their original resolution. Your driver can translate the resulting pixel map into printer codes at one screen pixel per printer pixel. The end result is a printed image with the same physical size as the original screen image but with a resolution twice as great.

Take a look at Figure 3. In Figure 3a, we show a circle and the letter A drawn at screen resolution. In Figure 3b, the same image is magnified, pixel by pixel, to about twice its normal size. It doesn't look any better, just bigger. However, if we have these objects in a picture, we can use `DrawPicture` to draw them at twice their normal size. The picture algorithm redraws the objects with increased resolution instead of simply magnifying existing pixels. The increased resolution allows QuickDraw to draw a much smoother circle (since the screen has the same resolution, but the circle has twice the radius) and a smoother-looking A since we use a 16 point font instead of an 8 point font. (Rather than drawing the font recorded in the picture and scaling the image, QuickDraw calls the Font Manager to get the best available font for the destination. Requesting a larger font size often returns a custom-designed font strike from disk, making a marked improvement in the appearance of text at higher resolutions.) The results of the picture scaling are shown in Figure 3c. Figure 3d shows Figure 3c scaled down to actual size.

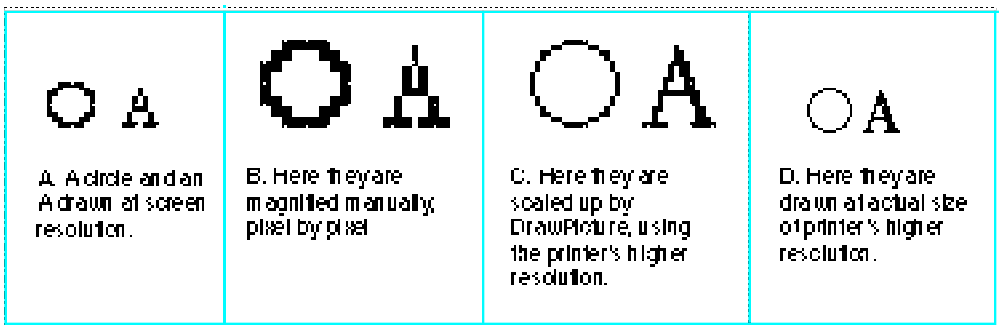

A. A circle and an A drawn at screen resolution.
B. Here they are magnified manually, pixel by pixel
C. Here they are scaled up by DrawPicture, using the printer's higher resolution.
D. Here they are drawn at actual size of printer's higher resolution.

Figure 3
A Demonstration of Picture Scaling vs. Magnification

Of course, if you want to draw objects at three times their normal size, you probably won't be able to draw an entire page at once. You can, however, draw them into a printing `GrafPort` with the clipping region set to a small rectangle of the picture. If you divide the page into ten such "bands," you only need one-tenth the memory the entire page would need. You just have to call `DrawPicture` ten times to complete printing for the page.

This technique is referred to as *banding* and is done by most printer drivers in deferred mode to work even in low-memory conditions. To image a full 8 1/2 by 11-inch page at three times resolution would require 506K per page (56K at normal resolution magnified by three horizontally and by three vertically), but dividing it into 20 bands requires only 25K per band—and the band buffer is reusable. Dividing it into 51 bands requires under 10K per band. Since applications are instructed not to call `PrPicFile` if a 10K buffer isn't available (see Volume 1 of the *Apple IIGS Toolbox Reference*, pages 15–30), you can always use a 10K buffer and you may be able to use a much larger one if memory is available. You'll have to divide it into 102 bands if vertical condensed mode is selected, since that doubles the vertical resolution.

The drawback to this method is that it's slow. QuickDraw can't know before interpreting the stored picture operations which ones will be clipped out and which will actually be drawn, so it spends a lot of time drawing the 50/51sts of the page that don't show up each time. If there are a lot of fonts on the page, the Font Manager spends time installing versions of them three times larger than the original, which in turn takes a lot of memory and makes things even slower. Generally, the more memory you can use for the band buffer, the faster printing will go. The fastest method would be to get the entire page imaged at once, but that's not always feasible.

WHAT YOU'LL NEED

The printer driver author has to create a set of routines that can accurately reproduce a graphic image on the printer or other reproduction device—FAX modem, graphic language device, and so on. Besides this article, you'll need information from a range of sources to write a good driver.

- **Apple IIGS Technical Notes #35 and #36.** Technical Note #35 is the *only* document that completely and authoritatively defines what each printer driver routine must do, as well as the structure for printer drivers. There have been mistakes in this note in the past. Since few developers have written printer drivers, we haven't gotten much feedback. This article was written using the September 1990 revision of the note, as well as corrections to the March 1990 version.
  Technical Note #36, Port Driver Specifications, is the complete specification for port drivers, listing the parameters for each call. The calls are made through the Tool Locator.
- ***Apple IIGS Toolbox Reference* series**, published by Addison-Wesley. The Print Manager and its data structures are defined in Volume 1; necessary QuickDraw routines are in Volume 2; and corrections and new calls to all the tools are in Volume 3. The beta drafts of any of these books are not good enough.
- **Knowledge of your target printing device.** If you can write a routine (in a desk accessory, perhaps) that can print a pixel map (like the entire screen), you have a good start for some of the imaging routines you'll need in your driver.
- **Familiarity with QuickDraw.** Since printing occurs when the application draws into a printing `GrafPort`, you have to be able to manipulate `GrafPorts` and their clipping components. To print in deferred mode, you have to be able to store images and reproduce parts of them for translation to printer codes.
- **Knowledge of the Print Manager architecture.** In addition to the 17 calls your driver will handle, you should be familiar with the other Print Manager and port driver calls so that you can use them to your advantage.

THE PHYSICAL STRUCTURE

There is a standard physical structure for printer drivers to follow so that the Print Manager can perform its dispatching properly.

COMPATIBILITY WITH APPLE'S DRIVERS

Apple's printer drivers have dominated the print driver development environment. This dominance has discouraged the creation of third-party drivers, which has in turn made a bad situation worse. Since there are few drivers other than Apple's to test with, applications tend to do unsavory things with drivers because they're expedient. Since applications do unsavory things, developers who want their drivers to be backward-compatible with applications tend to disassemble the Apple drivers to figure out what to do. Since everyone does unsavory things, the system winds up in an unusable and unmaintainable state because no one wants to rock the boat.

THE IMAGEWRITER DRIVER

Of all Apple's drivers, the old (pre-5.0.3) ImageWriter driver has caused the most headaches. The main problem with this driver is that it's a hybrid. Long ago, the structure of the Print Manager was quite different from what it is today. The Print Manager had "high-level drivers" and "low-level drivers." High-level drivers would communicate with the application, and low-level drivers would do the actual imaging or communication tasks. You will still see evidence of these things in some printing discussions. The giveaway is usually the abbreviations HLD for high-level driver and LLD for low-level driver.

When the Print Manager architecture was changed to its current design, the ImageWriter driver was converted—not rewritten as it should have been. The conversion created a lot of source files and put nearly every routine in a place where you wouldn't expect to find it. As new features were added, the entire thing became more and more unwieldy, until at last Ben Koning broke from the beast by creating a new, vastly improved ImageWriter driver for System Software 5.0.3 (with some imaging routines by Apple IIGS graphics wizard Jason Harper).

The Print Manager has a few features in it for the questionable use of the old ImageWriter driver. These have never been documented and now that the old ImageWriter driver is going away, these features may go away as well. If you have ever disassembled the driver (not kosher according to the license agreement anyway), you may have discovered some of these less-than-desirable programming practices:

- The ImageWriter driver used the value in the accumulator at entry time as a direct-page register.
- The ImageWriter driver tended to print correctly when print record fields were set to totally invalid parameters.
- Sometimes the ImageWriter driver played loose with the parameters. It was known to work acceptably when `printGrafPortPtrs` were passed where print record handles were expected.

WALKING A THIN LINE

All of this leads to the question of how you will write your driver: will you create your driver strictly by the book, or will you program defensively in an attempt to work with those who broke the rules? If you use only the defined print record fields and stay clear of undocumented structures, your driver will work fine with future versions of the Print Manager and most applications. On the other hand, if you don't support the unorthodox use of the print record, your driver is less likely to work with some of the bigger and more widely used Apple IIGS applications.

The scariest thing about continuing to support these structures is that it gives application authors no reason to stop using them. For practical reasons, it may be impossible to avoid using some of these undocumented structures. Keep in mind, however, that the less of this you can get away with, the better off everyone will be in the long run.

A printer driver begins with two zero bytes and a count of the number of routines the driver supports. The Print Manager will transform the call number into a precomputed index for a four-byte per entry jump table, and put this index in the `X` register. Thus an indirect indexed jump, `jmp(driverTable,X)`, will call the routine.

Note that each jump table entry is four bytes long, but a `jmp(driverTable,X)` instruction will only use the low word of each entry. This requires all your entry points to be in the same segment. To get around this, you can have a short entry segment that `JSL`s to routines in other segments. If you like, you can rewrite the entry code to use all four bytes of the address instead of the low two. Just remember to preserve the `X` register, as it's your only indication of which routine to call.

The entry point for the driver is at the fifth byte (just after the function count). Note that before September 1990, Technical Note #35 always had the table entries for `PrPixelMap` and `PrDriverVer` backward, and that `PrGetPgOrientation` was misspelled in the note. Also, the count of routines should be 17. A correct driver header looks like this:

```
DriverStart    START

               dc   i2'0'            ; identifying word
               dc   i2'(ListEnd-PrDriverList)/4'  ; count
EntryPoint     jmp  (PrDriverList,x)

PrDriverList   dc   a4'PrDefault'
               dc   a4'PrValidate'
               dc   a4'PrStlDialog'
               dc   a4'PrJobDialog'
               dc   a4'PrDriverVer'
               dc   a4'PrOpenDoc'
               dc   a4'PrCloseDoc'
               dc   a4'PrOpenPage'
               dc   a4'PrClosePage'
               dc   a4'PrPicFile'
               dc   a4'InvalidRoutine'
               dc   a4'PrError'
               dc   a4'PrSetError'
               dc   a4'GetDeviceName'
               dc   a4'PrPixelMap'
               dc   a4'PrGetPrinterSpecs'
               dc   a4'PrGetPgOrientation'
ListEnd anop
```

On entry to each routine, the stack looks just as it would for a Toolbox call. There are two `RTL` addresses, then any parameters, and finally any result spaces. The Print Manager dispatches to printer driver routines without adding any information to the stack, so you can imagine that the Tool Locator dispatches directly to your driver routine when a printer driver call is made.

Your entry code must be reentrant. Because the Print Manager will call some of your routines when you make port driver calls (like `GetDeviceName` when a port driver is first loaded), be sure you have no reentrancy problems.

The physical structure of printer drivers is the only constant thing about them. You can implement the rest of the driver in any way you choose, using resources, dynamic segments, and even multiple files. When you consider using other components like these, however, keep in mind that loading any of them may require users to insert the boot disk. Even if you make your resources have the `preload` attribute, most resources used by the system, like window and control templates, are released when the Toolbox is done with them. Marking them `preload` means the user won't have to insert the disk to use those resources the first time, but once they're released they're very likely to go away. You can get around this by loading the resources yourself and passing them to the Toolbox as handles instead of as resources—in which case `preload` resources work very well indeed.

THE LOGICAL STRUCTURE

In addition to the physical structure, there is a standard logical structure that printer drivers should follow so that printing actions are consistent from printer to printer. The driver consists of three functional parts: calls that do the printing loop, routines to maintain and access the print record, and other stuff—the few routines that don't fit either of the other categories.

PRINT LOOP ROUTINES

The printing routines will be called by the application to make printing happen. The application just opens a document, opens some pages, draws, closes the pages and the document, and when `PrPicFile` is called, printing just kind of happens. The printer driver is what makes it happen.

Although the printing routines are described fairly well in Technical Note #35, the following summary highlights the most important points about using these routines.

PrOpenDoc. `PrOpenDoc` is the beginning of the regular print loop. This is where you create (if necessary) and initialize the printing `GrafPort` for the application to draw pages into. You should also make sure to validate the print record, since it contains the settings you must use to image this document. If you want a "Preparing data" dialog box, this is the place to display it. Before you exit `PrOpenDoc`, you should have allocated most of the resources you'll need to print (memory, disk space, and so on).

PrOpenPage. `PrOpenPage` is the application's way of telling you "I'm going to draw into this `Grafport` to image the next page." You get to initialize the `Grafport` to be ready for printing, including setting the clipping regions to the size of the page rectangle (or the rectangle passed to `PrOpenDoc`, if there is one), and to make the printing `Grafport` the current one, saving the old port. If you're printing in immediate mode, you should install your bottleneck procedures in the `Grafport` here with the QuickDraw II call `SetGrafProcs`.

PrClosePage. `PrClosePage` undoes whatever it was that `PrOpenPage` did. Close the picture for this page here (or eject the page if you are printing in immediate mode). Be sure to restore the old `Grafport` (from `PrOpenPage`) before returning.

PrCloseDoc. `PrCloseDoc` similarly undoes what `PrOpenDoc` did. If `PrOpenDoc` allocated a new printing `Grafport`, `PrCloseDoc` must dispose of it (after making sure it's closed so you don't orphan any region handles). You should close the printing `Grafport` with the QuickDraw II call `ClosePort`. (It's not a port driver call, no matter what Note #35 says). You should also erase the dialog box you drew in `PrOpenDoc`, presuming you drew one.

PrPicFile. `PrPicFile` does nothing if you're in immediate mode, but it does nearly everything if you're in deferred mode. Given the model of recording pages in pictures, the instructions described in Note #35 are pretty good—they lead you through the process one step at a time.

There's one very important part of most printer drivers that's not covered by the note—imaging. The process of turning pixel images into printer codes is so dependent on the target device that neither this article nor the note can tell you how to do it. However, there are a few strategies that apply to all printer drivers:

- Doing fewer `DrawPicture` calls makes printing faster. The best way to do this is to use as large a band buffer as you can. Remember that `MaxBlock` doesn't reveal how much memory could be available after purging and out-of-memory routines, so just ask the Memory Manager for what you want, and ask for something smaller if you don't get it. Also remember to leave at least 16K available for the Toolbox and GS/OS : don't use all the available memory. See the Apple II Technical Notes for more memory management strategies.
- The conversion of pixels to printer codes will occupy most of your driver's executing time, so make it as efficient as possible. You should handle large areas of white space quickly by optimizing your conversion routines to scan for similarly colored areas as fast as possible.

- If your target device supports any kind of compaction or data compression, use it. Examples of compression include printer commands to “print this pattern 400 times” instead of sending “print this pattern” 400 times. Tests during Apple’s development of IIGS printer drivers have shown that even on a full page of text, the compression rate is always more than 50 percent.
- If you have any control over the port drivers, try to make them as fast as possible. Send data through the port driver in large chunks to let the port driver work as fast as possible. For a 300 dpi target device, there may be as much as one megabyte of data necessary to print all the pixels on an 8 1/2 by 11-inch page. Compaction will help here as well.

The status record is a method the application has of communicating with your printer driver, since printing can take such a long time. The job subrecord contains a pointer to a procedure to be called during idle time—that is, the time between pages, bands, or copies. If you’re passed `nil` for the `StatusRecPtr`, it’s probably easier for you to allocate a status record yourself and update it as if it were provided by the application.

Be sure to dispose of everything you’ve allocated during printing before leaving `PrPicFile`. Although the application should make all the print loop calls in order, if an error occurs inside one of the calls (or if the application calls `PrSetError`), the rest of the print loop must handle it gracefully and still deallocate all allocated resources at the end of `PrPicFile`.

PrPixelMap. `PrPixelMap` takes an arbitrary pixel map and prints it. You’re passed a QuickDraw `locInfo` structure (the pixel map defining portion of a `Grafport`), a rectangle enclosing the portion of the `Grafport` to print, and a flag indicating whether to use color. `PrPixelMap` is a quick and dirty way to print graphics without going through the print loop.

Your imaging code should have a routine to print an arbitrary pixel map anyway, and `PrPixelMap` can just call it. Alternatively, as suggested by Technical Note #35, you can allocate a new print record, make a picture that contains just the pixel map, and call your normal deferred printing routines.

PRINT RECORD METRICS ROUTINES

The print record metrics routines set and get values in the print record. The print record is the only way your driver can communicate with the application about printing parameters, making it vitally important that the print record be correct. Only you know if the values in the print record make sense, so you get to check it for consistency. You also get to present the most logical option choices to the user, since no one else knows what they are. In addition, there’s a new call for System Software 5.0 and later that lets you return the page orientation so that applications don’t have to go reading the print record.

PrDefault. This routine copies the default print record into the supplied handle. The default print record's contents will vary depending on the current screen resolution. Be sure not to set the handle size on this handle. Some applications keep extra stuff beyond the end of the record. This isn't kosher, but leaving the print record handle size unchanged is an easy work-around to a potential problem.

PrValidate. `PrValidate` checks a supplied print record for consistency. If any of the values are inconsistent or invalid, you should correct them. If the supplied print record isn't a print record from your driver, you should fill it with the default values.

PrStlDialog. `PrStlDialog` is responsible for the dialog box the user sees after choosing the Page Setup command in the File menu. You should initialize the controls in the dialog box based on the print record and save all the changes from the dialog box in the print record (if the OK button was pressed, of course).

PrJobDialog. `PrJobDialog` is responsible for the Print dialog box. As with the Page Setup dialog box, no one but your driver knows the best options and their default choices for your printer. `PrJobDialog` should initialize the `iCopies` field in the job subrecord to 1, `iFstPage` (the first page to be printed) to 1, and `iLstPage` (the last page to be printed) to the largest value your driver allows. By setting these values, you ensure that one copy of each page is printed if the user does not change these items. That's how the human interface should work.

PrGetPgOrientation. `PrGetPgOrientation` returns a 0 for portrait (small side on top) mode and a 1 for landscape (sideways) mode. No one cares where you store this in your print record, just return it here. For print records with `iDev` values $8001 and $8003, you must store this information in the `wDev` field.

MISCELLANEOUS DRIVER SUPPORT

There are a few routines involving port driver communication, printer identification, and internal functions that you get to provide as well.

PrError. You maintain an internal error code for your printer driver. This is so that if `PrOpenDoc` returns an error, you can look at the error code and do nothing for the rest of the print loop. `PrError` simply returns your internal error status.

PrSetError. `PrSetError` sets your internal error status to the supplied value. This call allows an application to clear an error state if it was able to resolve a specific problem.

GetDeviceName. GetDeviceName is also known as PrChanged—it's called by the Print Manager when your driver is first loaded. This routine takes the AppleTalk Name Binding Protocol or NBP-format name of your target device and passes it to the port driver routine PrDevPrChanged. This allows the network port driver to communicate with your target device over the network. If you don't have a network-compatible target device, pass nil to PrDevPrChanged. An example of an NBP-type name can be found in Picter.

PrDriverVer. PrDriverVer returns your driver's version number, so that applications can scope out your driver for features. If you document features that are available in a given version of your driver, this is how other code can find out if that version is here or not.

PrGetPrinterSpecs. PrGetPrinterSpecs tells the caller things about your driver without forcing any monkeying around with the print record. Your driver gets to return its iDev value identifying the kind of printer or style subrecord and the characteristics of the target device. Currently, the only defined characteristic is whether or not you're color capable. This stuff is defined for all existing iDev drivers, but it's good to keep people out of the print record anyway.

OUR SAMPLE DRIVER, PICTER

Picter is a very simple driver. It creates QuickDraw pictures of all the pages and saves them as picture files in the *:System:Drivers directory. (Picture files have the file type $C1, auxiliary type $0001.) The first file is named screen.a, and the last letter is incremented for each additional file until a pathname syntax error occurs.

Picter does not support many print record options. It prints only in color, portrait mode, full size. Picter has an `iDev` of $8001, so it interprets the style subrecord as the ImageWriter driver does. If someone sets a bit in the print record to an invalid value, Picter's `PrValidate` routine corrects it.

Picter is intended to be a working sample that shows the structure and content of a printer driver. It is a learning tool, not a release-quality utility. No printer driver with this many interface holes should see the light of day as a finished software product.

Picter is written in APW/ORCA assembly and uses the Make utility by 360 Microsystems for source code file management. If you don't have the Make utility, you can look in the make file to see the commands to build each of the components and the link order.

ABOUT BEN'S DIALOG BOX ROUTINES

Included with our sample printer driver are some dialog box routines from Ben Koning, the guy behind the new ImageWriter and ImageWriter LQ drivers. Ben has spent a lot of time creating drivers that are more powerful, faster, and easier to maintain so we can add more features in the future. Our thanks to Ben for sharing his routines, which have been slightly modified for use in this driver. If you ever see Ben around, buy him something really expensive—like a house, or a few cars, or a hot dog at the average trade show.

By using these routines, you can easily make your style, job, and status dialog boxes appear like those in Apple's printer drivers. Users will be less confused, everything will seem to fit together better, and the world will be a happier place.

There are two types of dialog boxes in these routines—status dialog boxes and interactive dialog boxes. The status routines make it very easy to keep the user informed during the printing process. There are three status dialog routines—one to display the empty dialog box, one to show a message in this box, and one to close the box:

- `StartStatusMessage` draws a small, blank dialog box centered on the screen, regardless of the mode (320 or 640).
- `StatusMessage` takes a pointer to a Pascal string in a direct-page location and displays that string centered in the status dialog box.
- `FinishStatusMessage` closes the dialog box and removes it from the screen.

Call `StartStatusMessage` at the beginning of `PrPicFile`, and every time you do something different, call `StatusMessage` with a descriptive string. Several descriptive strings are included as examples of what the new ImageWriter driver does. Call `FinishStatusMessage` before returning to the caller.

There are also two other specific dialog routines that our sample driver does not use. `StatusMesgFeedPrompt` fills the status dialog box with the string "Insert sheet for page: XXXXX", where you pass the page number necessary as an integer on direct page. `NotCorrectDevDialog` displays a small box with a Cancel button that indicates that this is not your target device.

`StatusMesgFeedPrompt` must be called between `StartStatusMessage` and `FinishStatusMessage`, but `NotCorrectDevDialog` can be called at any time.

The style and job dialog boxes are largely defined by the controls in them. `ConductStyleDialog` and `ConductJobDialog` each have predefined templates linked in as data. This way, you can avoid the disk-insertion problems that resources and dynamic segments entail. The item IDs are equated to match values in the print record. Picter shows how you can write the standard metrics routines to use Ben's dialog box routines.

THE WORLD ROUTINES

To ensure that our driver has a consistent environment, Picter includes a few environmental routines around every call and some at the main entry point.

Our entry point is the short, indirect jump, as we saw when we looked at the driver's physical structure earlier. This is acceptable because all of our entry points are in the same segment. Before making the jump, we call the environmental routine `MakeOurWorld`.

Because there are no printer driver startup and shutdown calls, some people have wondered how printer drivers can obtain direct-page space and release it. `MakeOurWorld` is a way to do this. It relies on the fact that when printer drivers are unloaded, they are not marked as restartable. Every time the driver is reloaded, we get a fresh copy of the driver from disk. So we link in a storage word of zeros, allocate our direct-page space, and store the address of this space in the zero word. Then on every entry, we just check that word. If it's zero, we were just loaded from disk, so we go get the direct-page space again. If the word isn't zero, it's our direct-page space: transferring it to the direct-page register after saving the current value sets our direct page.

We give the direct-page memory the same user ID as the driver. Thus when our driver is unloaded, the direct-page memory is likewise released.

If you don't need static direct-page space, by all means don't allocate any. If you use the application's existing stack frame instead of allocating the new direct-page space, you can conserve bank zero space. However, since allocating direct-page space is a little trickier, a solution is included in `MakeOurWorld`.

`MakeOurWorld` returns with the accumulator zero and the carry clear if everything was right. If the accumulator is zero and the carry is set, we were just loaded and our direct page is not initialized. If the accumulator is nonzero and the carry is set, there was a real error.

Immediately in every subroutine, Picter puts the number of bytes of input parameters in the `Y` register and calls `CheckTheWorld`. If there was a real error, `CheckTheWorld` calls `EndOurWorld` to get out of the printer driver with the error code. If there was no error, `CheckTheWorld` quickly returns to the caller.

`EndOurWorld` removes the saved values of the direct-page and data bank registers we pushed on the stack in `MakeOurWorld`. On entry, `X` contains an error code or the value $FFFF to indicate the internal error code should not be changed. The `Y` register contains the number of bytes of input parameters to pull. The routine that removes the input parameters is quite generic and is very similar to those used in the Toolbox's common exit routines.

PICTER'S METRICS ROUTINES

Because Picter is limited in its scope and abilities, its actual printer driver calls function slightly differently than they would in a full-blown printer driver. Here's a description of how Picter implements the standard print record metrics routines.

PrDefault. PrDefault does nothing more than copy a linked-in default print record to the handle passed as input. It then fixes the rPage and rPaper rectangles to match the current screen resolution.

PrValidate. PrValidate examines the print record values Picter knows about to make sure they match the values we support. If they don't, they are modified to be supportable and consistent.

PrStlDialog. PrStlDialog calls the ConductStyleDialog routine to do the actual Page Setup dialog box. The dialog routines call several very small subroutines in Picter to read the print record values. ConductStyleDialog never accesses the print record itself. This is an example of a method of print record management that I prefer.

PrJobDialog. PrJobDialog is very much like PrStlDialog in that it calls one of the dialog routines to conduct the dialog, and those routines call us for information on the print record.

PrGetPgOrientation. PrGetPgOrientation returns the value for page orientation out of the supplied print record. It reads the values directly, although it could call a metrics subroutine just as easily.

PICTER'S PRINT LOOP ROUTINES

These routines are Picter's implementation of the routines that do the actual printing.

PrOpenDoc. The actual print loop itself is also slightly unorthodox, due to the nature of the target device (QuickDraw picture files).

PrOpenDoc sets up a printing Grafport, validates the print record, and displays a small status message dialog box. It also initializes other printing parameters, like the internal error and page number variables.

PrOpenDoc stores variables on direct page, making it very bad if the driver were to become unloaded before PrPicFile. Since MakeOurWorld lets us check for this easily, we return a new error if it happens. The error is defined as $13FF and the equate is PrBozo. Any meaning this equate has is the interpretation of the reader.

PrOpenPage. PrOpenPage checks to make sure our direct page is still around and returns PrBozo if not. If all is well, we increment the page number and check the job subrecord to make sure this page is one we're supposed to be printing. If it is, we initialize the printing Grafport to contain rectangular clipping and visible regions the size of the rPage rectangle (or of the supplied frame rectangle, if any).

We update the status dialog box and call our subroutine OpenPICTFile, which creates the new picture file, opens it, and opens a QuickDraw picture for recording the page images.

PrClosePage. `PrClosePage` calls `ClosePICTFile`, which closes the picture, writes it to disk, and kills the picture. We then close the printing `Grafport`, update the status dialog box, and return. (None of this happens if the driver was just loaded. The caller gets `PrBozo` instead.)

PrCloseDoc. `PrCloseDoc` disposes of the memory for the printing `Grafport` if it was allocated by `PrOpenDoc`. We restore the old `Grafport`, close the status dialog box, and exit.

PrPicFile. `PrPicFile` doesn't really do anything in Picter. We do all our actual "printing" in the page routines, but our job record indicates that we are in deferred mode for compatibility with applications that don't think they print in immediate mode. Nevertheless, Picter shows how to do several of the more common `PrPicFile` actions, like setting up a status record, allocating and initializing a new `Grafport` for imaging the pages, calling the idle procedure in the job subrecord, and displaying the status dialog box.

PICTER'S MISCELLANEOUS ROUTINES

These routines are Picter's implementation of the routines that make your driver complete. They allow your driver to respond to requests for error, network, and version information.

PrReserved. `PrReserved` is the name we picked for what Note #35 calls `InvalidRoutine`. It is, in fact, the remnants of an old Print Manager architecture call named `PrControl`. This had varying parameters and was generally not Your Friend. To be safe here, we return error $0002, which as a Tool Locator error indicates to the caller that he should pull his parameters back off the stack.

PrError and PrSetError. `PrError` returns the value in our internal direct-page error location. `PrSetError` takes the value and puts it in our error location on direct page.

GetDeviceName (PrChanged). `GetDeviceName` really has no meaning for us, since our target device doesn't (and can't) exist on an AppleTalk network, but an NBP-type string is included anyway to demonstrate the technique. This will cause the network port driver to report that no devices of our type are available.

PrDriverVer. `PrDriverVer` returns the version word for our driver. You might want to stop in the middle of writing your `PrPicFile` call to write `PrError`, `PrSetError`, and `PrDriverVer` just to remind yourself that it's not always that hard.

PrGetPrinterSpecs. `PrGetPrinterSpecs` returns our `iDev` word and the color capabilities of this printer (picture files are always in color). If you need to check your target device's capabilities (for example, an ImageWriter doesn't always have a color ribbon in it), this is the place to do it.

WHAT YOU CAN ADD

Picter is intended as a workbook, a shell from which you can learn printer driver technique. There are many more things you can do with it before starting your own printer driver. By examining these areas now—before you actually try to implement them in a driver—you will avoid future frustration.

More picture types. Picter writes only QuickDraw picture files as supplied. You could add a pop-up "Picture type" menu to the job dialog box and allow the user to pick any of the popular graphics formats. Apple Preferred is a good choice because its line-oriented structure makes it a good candidate for banding. Banding will be necessary unless you have a pixel map large enough to hold the entire image at once. Other easy additions are packed QuickDraw pictures and 32K screen dumps (if you can get a 32K block for the pixel map). Remember that screen files aren't 32K of pixels—they're 32,000 bytes of pixels and 768 bytes of scan-line control bytes and color tables.

More page types. As supplied, Picter only supports two types of page metrics—screen size and U.S. letter size. Try adding more sizes (legal, label, envelope). The code to handle different page metrics is directly applicable to any other printer driver. In fact, you could add line edit controls to let the user type the size of the page rectangle in inches or centimeters and thus have no limit to the number of paper sizes you support.

Communicating with the port driver. Picter doesn't communicate with the port driver (except in `GetDeviceName`). Try writing the name of each call to the port driver as it executes. If you have an ASCII printer connected to the hardware controlled by the port driver, you should get a hard copy of each call name as it executes. You could also write debugging information this way, such as parameters or print record addresses.

More options. You can also add more standard print record options—such as condensed and landscape modes—to Picter. Supporting landscape mode involves swapping the horizontal and vertical coordinates of the `rPage` and `rPaper` rectangles as well as the horizontal and vertical printer resolutions—just be sure your validation routines know how to deal with it! You can make vertical condensed mode happen by passing a rectangle that is half the correct height of the framing rectangle for `OpenPicture`. Other reduction values, both horizontal and vertical, come by changing the framing rectangle for `DrawPicture` as well.

For More Information
Apple IIGS Toolbox Reference, Volumes 1–3
Apple IIGS Technical Note #34, Low-Level QuickDraw II Routines
Apple IIGS Technical Note #35, Printer Driver Specifications
Apple IIGS Technical Note #36, Port Driver Specifications
Apple IIGS Technical Note #51, How to Avoid Running Out of Memory
Apple IIGS Technical Note #72, QuickDraw II Quirks
Apple IIGS Technical Note #93, Compatible Printing •

GO FORTH AND IMAGE

Printing doesn't have to be a big mystery. The task is divided into components so that no one part of it becomes insurmountable. Turning imaging into printer codes is the responsibility of the printer driver, talking to the hardware is the responsibility of the port driver, and the Print Manager holds it all together. While supporting different printers and interfaces would normally be beyond the scope of most applications, the Apple IIGS printing architecture makes it easy for applications. All you need is a printer driver—and now you know how to create those as well.

Thanks to Our Technical Reviewers
Pete "Luke" Alexander, Ben Koning, Suki Lee, Jim Luther, and Dave Lyons •

INDEX

This is a cumulative index. All page numbers for articles in this issue are in color and all page numbers for articles in previous issues are in black. The first issue ran from pages 1-112, the second issue ran from pages 113-256, the third issue ran from pages 257-372, and this issue runs from pages 373-500. For a complete source code listing, see the *Developer Essentials* disc. •