COMPATIBILITY: RULES OF THE ROAD

Apple's System Software Version 7.0 provides the most important test of compatibility since the introduction of the Macintosh II. This article should help you prepare for the release of System 7.0. For an overview of the most critical compatibility issues and how to address them, read on.

If you've already read too many stuffy articles full of dire warnings about compatibility, you've probably decided this one will be best suited for lining the bottom of your filing cabinet. But before doing that, consider the case of Johnny Appledweeb.

Ace Macintosh programmer for Cliff Grazer Enterprises, Johnny is currently putting in the long hours to get a spread-processor-terminal-graphics-emulator out the door. He doesn't have time to read an article like this because Cliff Grazer, his boss and President of CGE, is all over his case. Four months ago, the company began accepting prepayment from customers who can't wait for Johnny's program to reach their local stores. Those customers are now beating down the doors.

Although Cliff is desperate to get the product out, he has required certain levels of performance. The application must be kept under the 1megabyte limit, for example, and must keep up at 19.2 kbaud through the modem port. Finally, at the last minute, legal decides to require copy protection. Once the application ships, reviews are excellent, customers are happy, and sales are good. Cliff is ecstatic and gives Johnny a big raise. Johnny has time to relax a bit and maybe even catch up on some reading. But this article doesn't interest him because he's a crack programmer, and his application works fine. Bottom of the filing cabinet time.

Six months later, Johnny comes back from his well-earned vacation to find that Apple has introduced new machines and released a new version of the system software. Cliff's hopping mad because of reports of compatibility problems and complaints from angry customers. As Johnny begins to look into the problems, he has a vague recollection of some article he saw on compatibility. He rummages around, finds the article, and quickly discovers it addresses his problems. But it's too late for the customers. They don't understand compatibility, but they do understand that the application they have been using every day no longer works. Cliff doesn't

DAVE RADCLIFFE, "Technical Sherpa," has been with Apple about a year and a half, putting his chemistry degree from Washington University to work in A/UX® and MPW™ technical support. Actually, he discovered his true calling while working with the computers in the UCLA chemistry research labs. When asked how he's changing the world one person at a

really understand compatibility either. What he understands is he now has the expense of shipping updated versions to keep his customers happy.

Johnny might have saved himself and others a lot of trouble if he'd spent a few minutes with this article right away. Sure, it probably would have meant a delay in the first release of the application, but it might also have made a second release unnecessary. Johnny's a good programmer, and he's aware of almost everything in this article, but if a single sentence had helped avoid problems, the article would have been worth Johnny's time.

It may be worth your time as well to check out the compatibility of your current application's features with System 7.0. The road gets a little dry and dusty from here on, so grab a cold one and we'll get down to business. This article focuses on specific areas of Macintosh programming where compatibility might trip you up today or in the future. It isn't meant to be a guide to Macintosh programming, so if you need additional information on a topic, such as implementation details, refer to *Inside Macintosh*, volumes I-V, and the Macintosh Technical Notes.

DEFENSIVE PROGRAMMING

Murphy was clearly a computer engineer. If anything can go wrong with your application, it will, as most of us learn the hard way. Once you recognize that users always stress your program in ways you never thought possible, you acquire defensive programming habits.

TESTING

Always test return values for possible errors because you never know when some unusual situation will arise. Assume that data structures will change. The Memory Manager is an example of a manager whose data structures are changing, as described later in this article. Avoid any portion of a data structure marked "Unused"—its use is reserved for Apple.

MEMORY ALLOCATION

If you treat the Memory Manager with a little courtesy and respect, your application will live a long and happy life. Keep in mind the strengths as well as the limitations of the Memory Manager and listen to what it tells you. Believe it when it returns a nil handle to tell you of memory allocation failure. Every application's memory needs are different, and as you design your application, think about how memory you allocate will be used. A little planning can ease the Memory Manager's task by reducing the number of Memory Manager calls and minimizing fragmentation and thrashing.

You should ask yourself a few simple questions about the memory you allocate in the heap. Is this memory you will need frequently? Rather than frequently allocating and releasing the memory, wouldn't it be better to allocate it once at the start of your

time, Dave replied, "home-brewed beer." In addition to home concoctions, he's into hiking, backpacking, and photography. •

application; if it is a handle, move it high in the heap with MoveHHi; and simply reuse it when necessary?

Is it memory that shouldn't move? If so, consider the use of NewPtr instead.

Is this a large block of memory used for a very short period of time? Judicious use of MultiFinder temporary memory can satisfy such needs and reduce overall heap usage, allowing you to shrink your MultiFinder size partition.

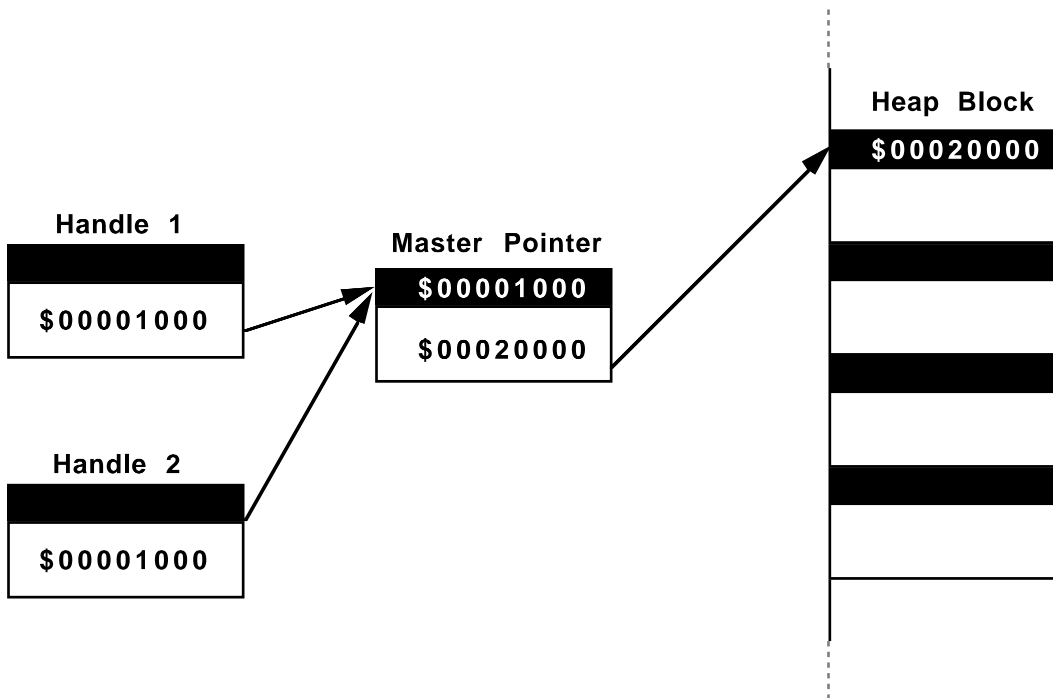
Is this memory you are willing to let the Memory Manager dispose of at its discretion, such as for a resource? Then you should consider making it purgeable. But if you've made it purgeable, be sure to check for empty handles.

Once you have your application working, be sure to stress test your use of the Memory Manager. You can do this by using your debugger to force heap scrambling and purging. You can also simulate low memory conditions by running your application in a small MultiFinder partition.

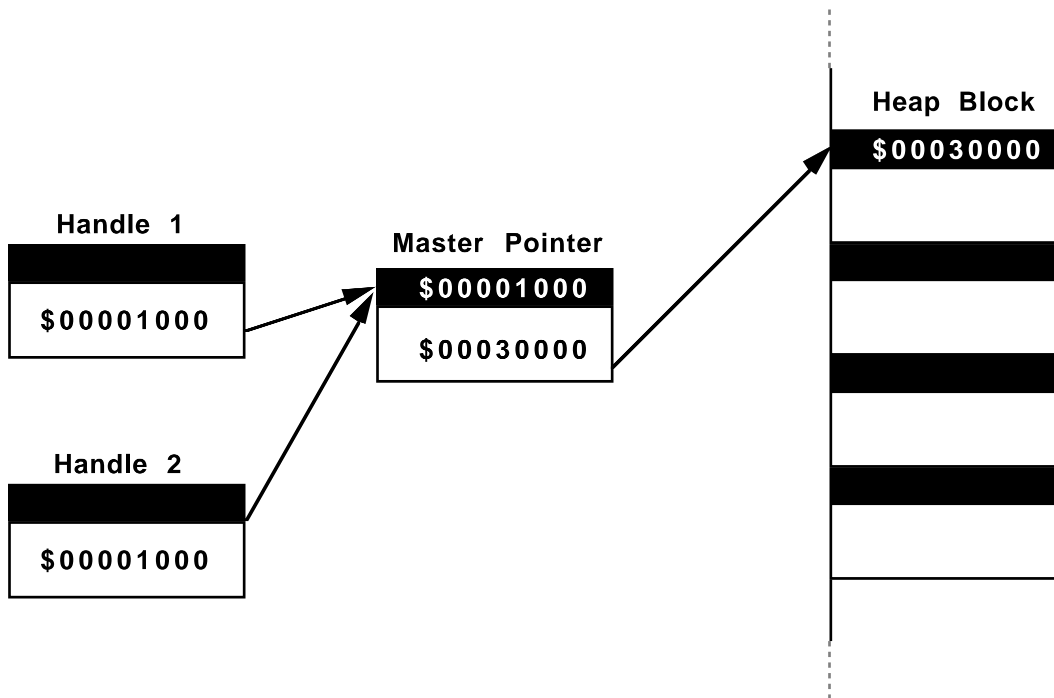
32-BIT CLEANLINESS

Another way to treat the Memory Manager with kindness and respect is to practice 32-bit cleanliness. Being 32-bit clean may be the single most important compatibility issue facing developers. To understand what 32-bit cleanliness means, let's take a closer look at Macintosh memory management. The Memory Manager maintains free-form memory structures called heap zones. It allocates memory blocks of various sizes within these zones to satisfy memory allocation requests by the system and applications. Occasionally, heaps will become full or fragmented and the Memory Manager will need to rearrange or purge blocks in a zone to create enough contiguous space to satisfy a memory allocation request. To minimize confusion that could occur when blocks are rearranged, the Memory Manager uses indirect references called handles to refer to relocatable blocks in the heap.

The Memory Manager maintains a series of master pointers referring to blocks in memory. A handle is a pointer to a master pointer, as shown in the following illustrations.

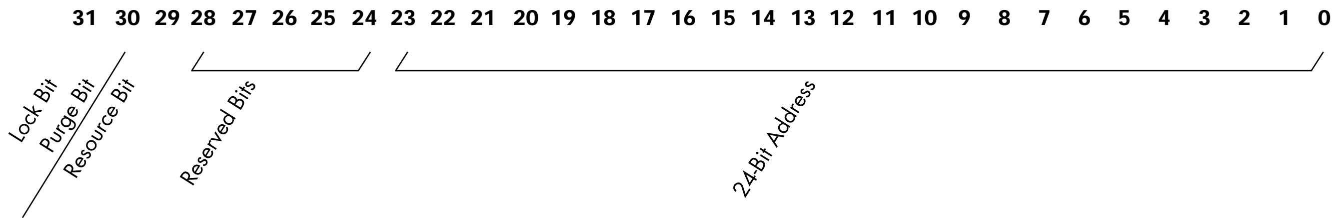


In the example in the first illustration, two independent handles refer to the same heap block at address $20000 via the master pointer at address $1000. Only the master pointer should be referring to the heap block. Now, suppose the system needs to relocate the heap block to address $30000. The second illustration shows the state of the system after relocating the block.



The master pointer is now correctly set to point to the new block. The master pointer is the only thing the Memory Manager had to update. The original handles 1 and 2 still correctly refer to the heap block because they refer to the master pointer, which has the correct location of the heap block.

The classic Macintosh has what is referred to as a 24-bit memory management system. To the hardware, only the lower 24 bits of a 32-bit address are significant. The upper 8 bits are always ignored in a hardware address reference. The Memory Manager maintains certain information about heap blocks, such as whether they are locked in memory and cannot be moved or whether they can be purged from memory to free up space in the heap. The original Macintosh Memory Manager took advantage of the unused upper 8 bits of the address in a master pointer to maintain flags about heap blocks. The illustration shows the master pointer structure of the 24-bit Memory Manager.



AVOIDING COMMON PROBLEMS

The most common violation of 32-bit cleanliness involves direct manipulation of Memory Manager flags. In a 32-bit system, all 32 bits of an address are valid, and in the case of a master pointer, the flags bits are stored elsewhere. The system provides traps for setting and cleaning these flags: `HLock/HUnlock`, `HPurge/HNoPurge` and `HSetRBit/HClrRBit`. There are also traps for getting and setting all the flags at once: `HGetState` and `HSetState`. Some applications have taken advantage of knowledge of the master pointer structure to set and clear the flag bits directly. Setting flag bits directly on a 32-bit system means you are not changing the flags, but changing the address itself, and suddenly your master pointer is pointing to a completely different location in memory.

The issue of 32-bit cleanliness is not limited to proper use of master pointer flags. Every address reference must assume all 32-bits are valid. If you have used any of the upper 8 bits of pointers or handles for anything other than part of an address, you must find an alternate representation for that information.

Two other places you can be bitten by 32-bit violations are in window definition functions (WDEFs) and control definition functions (CDEFs). The original Macintosh Window Manager stored the window variation code in the upper 8 bits of the handle to the window definition procedure. If you are using custom WDEFs and need to access the window variation code, use the `GetWVariant` trap. Similarly, use `GetCVariant` to retrieve the variant control value for a control that was formerly stored in the high bits of the control defproc handle.

Using pre-System 7.0 software, including A/UX 1.1, it is impossible to write a strictly clean CDEF. The problem with custom CDEFs is that the `calcCRgns` message uses the high bit of the region handle as a flag. *Inside Macintosh*, volume I, page 331 incorrectly advises you to “clear the high byte (not just the high bit) of the region handle before attempting to update the region.” Rather, you should clear only the high bit (not the high byte). This makes the reasonable assumption, given the current system software, that the handle represents only a 31-bit address and clearing the high bit is not harmful.

With System 7.0, the Control Manager has a new way of telling your CDEF to calculate control regions. Two new messages have been defined, `calcCntlRgn` and `calcThumbRgn`, with values of 10 and 11 respectively. With a 32-bit Memory Manager in operation, the Control Manager, which previously would have used `calcCRgns`, will now use one of the new messages. With a 24-bit Memory Manager operating, `calcCRgns` will still be used, so you must continue to support that method.

CREATING VALID HANDLES

Just as the master pointer structure will change in System 7.0, other Memory Manager structures will be subject to change. As a precaution, you should not access Memory Manager data structures directly or attempt to "walk the heap" yourself.

Since a handle is a pointer to a pointer, it is possible for an application to create a handle itself, a so-called fake handle. If you pass a fake handle to any Memory Manager routine, the Memory Manager will assume it is a valid handle under its control and may try to relocate or dispose of it. You should never pass a fake handle to any Macintosh trap, because you never know when that trap may itself call the Memory Manager.

Prior to System 7.0, handles allocated with `MFTempNewHandle` trap were not true handles and could not be passed, directly or indirectly, to Memory Manager traps. They were to be treated as fake handles. Under System 7.0, this is no longer true; the Memory Manager knows how to manage such memory.

Remember that MultiFinder temporary memory is just that, temporary. It should be allocated, used, and released as quickly as possible, preferably within one event loop cycle. With System 7.0, you can use the `HPurge` Memory Manager trap to mark handles as purgeable. You can continue to use the memory as long as MultiFinder does not need it for another application. But be sure to check for empty handles to ascertain if your memory has been purged.

USING STRIPADDRESS

One of the keys to 32-bit cleanliness is proper use of the `StripAddress` trap. `StripAddress` is necessary because handle flags in master pointers can create dirty address references. When a 24-bit Memory Manager is operating, `StripAddress` clears the high byte of the address, and returns a clean address. The operation of `StripAddress` is simple enough. What is not always so clear is when use of `StripAddress` is necessary or even appropriate.

To understand the operation of StripAddress, consider, again, the second illustration. Imagine that a 24-bit Memory Manager is in operation and you've called HLock to lock the handle. The value of the master pointer will now be $80030000 because HLock has set the lock bit in the master pointer as indicated in the third illustration. In normal operation, you never need to concern yourself with that high byte because the hardware ignores it. In other words, the hardware quietly strips the address for you. But suppose you are writing a driver that needs to access a NuBus board. To do that, you need to switch the hardware to 32-bit addressing mode using SwapMMUMode. Now, suddenly, the hardware is no longer ignoring that high byte, so to access the address properly you first need to call StripAddress to clean up the address.

Another situation in which StripAddress is necessary is comparing master pointers. In the previous example, comparing the value of the master pointer before and after calling HLock would lead you to conclude the master pointer is now pointing to a different block because the comparison looks at all 32 bits. To be sure you are comparing the relevant portions, namely the addresses, call StripAddress before comparing master pointers.

If a 32-bit Memory Manager is in operation, StripAddress will return the address unchanged, because all 32 bits of the address are valid. If you have used StripAddress correctly, you need never worry whether a 24-bit or 32-bit Memory Manager is operating, because StripAddress does the right thing.

Finally, do you need to call StripAddress on other addresses, such as handles? No, because there should be no extraneous bits set in the high byte of handles. If you are using the high byte of handles for your own purposes, go directly to the beginning of this section on 32-bit cleanliness. Do not pass Go; do not collect $200.

FILE ACCESS

Use the File Manager for all your file access. Avoid assumptions about the underlying file and directory structure. Not only has the Macintosh file system changed in the past, but you might not even be accessing a Macintosh volume. Foreign file systems such as DOS, ProDOS®, High Sierra ISO 9660, and Unix are all supported. If your application is running under A/UX, there may be no Macintosh volumes. These file system differences create many subtle problems. For example, Unix filenames are case-sensitive, whereas Macintosh names are not. Unix uses '/' as a pathname delimiter, while Macintosh uses ':'. Different file systems may have different restrictions on the length of filenames. Always use SFGetFile and SFPutFile. Not only will this ensure maximum compatibility across file systems, but it will be comforting to your users that your application looks and behaves like other Macintosh applications.

PRINTING

Apple is working to make printing easier for Macintosh programmers in the near future. Meanwhile, we can offer some help in two areas that often cause problems when printing: handling print records and using PostScript.

HANDLING PRINT RECORDS

Some applications set fields in the print record to change the default settings of items in the print dialogs. Rather than modify these fields, applications should just save the print record after the user has configured it. The best method for saving the record is to save it as a resource in your document's resource fork. Since a valid handle already points to the print record, creating a resource is easy:

```
/* This is an example of saving a print record into a resource file.  Saving the      */
/* print record in document resource files provides a method of retaining the         */
/* user setting from the last print job. For example, if a user elects to print a     */
/* document using landscape orientation, that information is stored in the print      */
/* record. If the record is saved with the document, the orientation information      */
/* will be available for the next time the document is printed.  When the 'Page      */
/* Setup' dialog is presented, the user's choices from the last time the document     */
/* was printed will be displayed as defaults.  This provides a convenient, device     */
/* independent method for saving print job information.                              */
/*  NOTE: Information from the Page Setup dialog is saved into the print record.      */
/*  Information from the Print dialog (i.e. # of copies, page range...) is            */
/*  considered to be per job information, and is not saved.  This method              */
/*  will not allow you to provide new defaults for the PrJobDialog.                   */
/*                                                                                    */
/*  Version              When          Who           What                             */
/*  1.0                  7/18/89       Zz            First release.                   */
/*                                                                                    */
```

```
#include <Values.h>
#include <Types.h>
#include <Resources.h>
#include <QuickDraw.h>
#include <Fonts.h>
#include <Events.h>
#include <Windows.h>
#include <Menus.h>
#include <TextEdit.h>
#include <Dialogs.h>
#include <Desk.h>
#include <ToolUtils.h>
#include <Memory.h>
#include <SegLoad.h>
#include <Files.h>
#include <OSUtils.h>
#include <OSEvents.h>
#include <DiskInit.h>
#include <Packages.h>
#include <Printing.h>
#include <Traps.h>

/* POPT = Print OPTions.  This type can be anything   */
/* but to avoid confusion with Printing Manager       */
/* resources, the following types should NOT be       */
/* used: PREC, PDEF, & POST...                  */
#define gPRResType 'POPT'

/* This can also be any value.  Since there should    */
/* only be one print record per document, the ID is   */
/* a constant.                    */
#define gPRResID   128

/* Resource name. */
#define gPRResName "\pPrint Record"

/* Define the globals for this program... */
THPrint gPrintRecordHdl;
short   gTargetResFile;
```

```
/* ReportError                         */
/*                                     */
/* This procedure is responsible for reporting an error to the user.  This is     */
/* done by first converting the error code passed in theError into a message      */
/* that  can be displayed for the user.  See Technical Note #161, "When to call   */
/* PrOpen and PrClose".                                                            */
void ReportError(theError)
OSErr theError;
{
   /* Real programs handle errors by displayed comprehensible error messages.     */
   /* This is NOT a real program...                                                */
   if (theError != noErr)
      SysBeep(10);
}

/* InitializePrintRecord                                                           */
/*                                                                                 */
/* This procedure is responsible for initializing a newly created print record.   */
/* It begins by calling PrintDefault to fill in default values, and then presents */
/* the standard 'Page Setup' dialog allowing the user to specify page setup       */
/* options.  The modified print record is then returned.                           */
void InitializePrintRecord(thePrintRecord)
THPrint thePrintRecord;
{
   Boolean     ignored;

   PrOpen();
   if (PrError() == noErr) {
      PrintDefault(thePrintRecord);
      ignored = PrStlDialog(thePrintRecord);
   }
   PrClose();
}
/* SavePrintRecord                                                                 */
/*                                                                                 */
/*  This procedure is responsible for saving a print record into a resource file. */
/* On entry, the print record should be initialized, and the resource file should */
/* be open with permission to write.                                               */
```

```
void SavePrintRecord(thePrintRecord, theResFile)
THPrint thePrintRecord;
short theResFile;
{
   short   currentResFile;
   Handle  existingResHdl;
   Handle  newResHdl;
   OSErr   theError;

   /* First save the currently selected resource file (before calling UseResFile).*/
   currentResFile = CurResFile();

   /* Now select the target resource file.                                         */
   UseResFile(theResFile);
   theError = ResError();
   if (theError == noErr) {
      existingResHdl = GetResource(gPRResType, gPRResID);
      if (existingResHdl != NULL) {
   /* There is already a print record resource in this file, so we need to         */
   /* delete it before adding the new one.                                         */
         RmveResource(existingResHdl);
         theError = ResError();
         if (theError == noErr) {
         /* If the resource was successfully removed, dispose of its memory        */
         /* and update the resource file.                                          */
           DisposHandle(existingResHdl);
           UpdateResFile(theResFile);
         }
      }
      if (theError == noErr) {
      /* Okay, now we have successfully opened the file, and deleted any           */
      /* previously saved print record resources.  Finally we can add the new      */
      /* one...                                                                    */
      /* Since the Resource Manager is going to keep the handle we pass it,        */
      /* we need to make a copy before calling AddResource.  We'll let the         */
      /* system do it for us by calling HandToHand.                                */
```

```
         newResHdl = (Handle)thePrintRecord;
         theError = HandToHand(&newResHdl);
         if (theError == noErr) {
           AddResource(newResHdl, gPRResType, gPRResID, gPRResName);
           theError = ResError();
           if (theError == noErr)
                UpdateResFile(theResFile);
           theError = ResError();
         }
      }
   }
   if (theError != noErr)
      ReportError(theError);

   /* Be polite and restore the original resource file to the top of the chain.   */
   UseResFile(currentResFile);
}


/* GetPrintRecord                                                                  */
/*                                                                                 */
/*  This function is responsible for loading a resource containing a valid print   */
/* record.  On entry theResFile should be open with permission to read.           */
THPrint GetPrintRecord(theResFile)
short theResFile;
{
   short   currentResFile;
   Handle  theResource;
   OSErr   theError;
   currentResFile = CurResFile();
   UseResFile(theResFile);
   theError = ResError();
   if (theError == noErr) {
      theResource = GetResource(gPRResType, gPRResID);
      theError = ResError();
```

```
      if (theError == noErr) {
        PrOpen();
        theError = PrError();
        if (theError == noErr) {
          if (PrValidate((THPrint)theResource)) ;
        }
        PrClose();
      }
   }
   if (theError != noErr)
      ReportError(theError);
   UseResFile(currentResFile);
   return((THPrint)theResource);
}
/* TestPrintRecord                                                              */
/*                                                                              */
/*  This procedure is used to test a print record.  It will print a line of text  */
/* using the options specified in thePrintRecord passed.  On exit, a line of text */
/* will have been printed.                                                      */
void TestPrintRecord(thePrintRecord)
THPrint thePrintRecord;
{
   GrafPtr     currentPort;
   TPPrPort    thePMPort;
   OSErr   theError;
   TPrStatus   thePMStatus;
   GetPort(&currentPort);
   PrOpen();
   if (PrError() == noErr) {
      if (PrJobDialog(thePrintRecord)) {
        thePMPort = PrOpenDoc(thePrintRecord, NULL, NULL);
        if (PrError() == noErr) {
          PrOpenPage(thePMPort, NULL);
          if (PrError() == noErr) {
               SetPort(&thePMPort->gPort);

               MoveTo(100, 100);
               DrawString("\pThis is a test...");
```

```
        }
          PrClosePage(thePMPort);
        }
        PrCloseDoc(thePMPort);
        if (((*thePrintRecord)->prJob.bJDocLoop == bSpoolLoop) && (PrError() == noErr))
          PrPicFile(thePrintRecord, NULL, NULL, NULL, &thePMStatus);
      }
    }
    theError = PrError();                /* Any errors?                          */
    PrClose();          /* Close the Printing Manager before attempting */
                /* to report the error.          */
    if (theError != noErr)               /* If there was an error during printing...*/
       ReportError(theError);            /* ...report the error to the user.       */
    SetPort(currentPort);
}
main()
{
    InitGraf(&qd.thePort);
    InitFonts();
    InitWindows();
    InitMenus();
    TEInit();
    InitDialogs(NULL);
    InitCursor();

    /* Get the ID of our resource file.  Since we were just opened, the          */
    /* CurResFile() will be ours.  In a real application, the resource file ID   */
    /* would be the ID of your application's document file.                      */
    gTargetResFile = CurResFile();

    /* Create a valid print record                                                 */
    gPrintRecordHdl = (THPrint)NewHandle(sizeof(TPrint));
    if (gPrintRecordHdl != NULL) {
       /* Okay, we got a print record, now initialize it.                         */
       InitializePrintRecord(gPrintRecordHdl);
```

```
        /* Now save the print record into the resource file.     */
        SavePrintRecord(gPrintRecordHdl, gTargetResFile);

        /* Now that it's saved, kill it off.  We'll restore it by  */
        /* calling GetPrintRecord.                                   */
        DisposHandle((Handle)gPrintRecordHdl);
        gPrintRecordHdl = NULL;

        /* Now get the print record from the file.  Since the     */
        /* record will be loaded as a resource handle anyway, let  */
        /* GetPrintRecord allocate the handle.                       */
        gPrintRecordHdl = GetPrintRecord(gTargetResFile);
        if (gPrintRecordHdl != NULL) {
           /* Now use the print record to see if the information we */
           /* saved was preserved...                                 */
           TestPrintRecord(gPrintRecordHdl);
        } else
           ReportError(MemError());
    } else
        ReportError(MemError());

    /* Kill the print record (if it was created) and go home...   */
    if (gPrintRecordHdl != NULL)
        DisposHandle((Handle)gPrintRecordHdl);
}
```

There are several points to remember when using this technique. Use a resource type not used by the Printing Manager so it doesn't become confused. Types to avoid include 'PREC', 'PDEF' and 'POST'. Remember that lowercase resource types are reserved for use by Apple. You also should not make assumptions about the size of the record. Use `GetHandleSize` if you really need to know. This allows for the record to grow in size in the future. Finally, when rereading the record from your document, be sure to pass it to `PrValidate` before using it in case the user has changed printers or print drivers since last printing the document.

USING POSTSCRIPT

Some applications prefer to bypass QuickDraw and print using PostScript instead. This often results in poor or nonexistent support for printers such as the ImageWriter and LaserWriter II SC. It also means relying on a method for determining which printer is in use, such as checking the `wDev` field in the `TPrStl` record.

One method for printing PostScript without relying on the type of printer being used is using the `TextIsPostScript PicComment`:

```
PicComment (PostScriptBegin, 0, NIL);
PicComment (TextIsPostScript, 0, NIL);
DrawString (ThePostScript);
PicComment (PostScriptEnd, 0, NIL);
```

The problem with this technique is that because non-PostScript printers ignore the TextIsPostScript PicComment, `DrawString`, which is a QuickDraw procedure, literally sends `ThePostScript` to the printer, resulting in garbage being printed. A better technique is using the `PostScriptHandle PicComment`. Because this comment is only understood by PostScript drivers, it avoids the QuickDraw / PostScript interaction just described:

```
PicComment (PostScriptBegin, 0, NIL);
PicComment (PostScriptHandle, GetHandleSize (ThePostScript),
    ThePostScript);
PicComment (PostScriptEnd, 0, NIL);
```

Further problems occur with applications that never print using QuickDraw but only use PostScript. Some versions of the LaserWriter® driver assume that if they see no QuickDraw, nothing was printed on the page and no output occurs. This can be avoided by embedding some nonprinting QuickDraw in your code. Immediately after calling `PrOpenPage`, issue the following calls:

```
PenSize (0,0);
MoveTo (10, 10);
Line (0,0);
PenSize (1,1);
```

This technique also solves a problem with background printing. In this case, the Printing Manager starts off each page with an empty default clipping region. Without seeing any valid QuickDraw calls, this region is never altered and your nice PostScript output is clipped entirely off the page. For more details on printing, see the article on “The Perils of PostScript” in this issue.

FONTS

System 7.0 will introduce an alternate way of dealing with fonts. While this new technology won’t cause problems for most applications, you should be aware of a few issues. Any application that allows user font selection will be affected by the new outline font technology. The most obvious feature is that any size font is now available. That means a list of point sizes in a menu is no longer sufficient. If you currently combine font selection and font size selection in a dialog box, be sure to include an editable field that allows the user to type in any point size. If you now

have a list of common sizes in a menu, include an "Other…" menu item that displays a similar dialog box with an editable field.

Since Apple introduced the LaserWriter, there has been a problem about where to get font metrics. The most compatible method is simply to call `FontMetrics` and read the metrics from the width table. For one reason or another, however, applications have seen fit to read metrics directly from the '`FOND`' resource. The addition of outline fonts adds another layer of complexity. Outline fonts will store metric information in the '`sfnt`'. Accessing metrics in the '`FOND`' could give invalid data. If you are currently accessing the '`FOND`' directly, you will have to revise to take advantage of '`sfnt`'s.

INTERNATIONAL SUPPORT

You can greatly expand the market for your product if you do not make assumptions about your user's language. Following a few simple rules can make your application much easier to localize. Don't simply assume, like many C programmers, that a character is one byte. Using the C routine `strcmp`, for example, to sort strings can give completely wrong results in languages other than English. Use `IUCompStr` instead. Determine the local conventions for decimal point, thousands separator, list separator, and time cycle from the appropriate international resource when performing input and output. Script Manager 2.0 routines, if available, can make this even easier by doing the right thing for you automatically. For example, the `Str2Format` routine can take input in one language and convert it to a canonical form that can be used by `Format2Str` to output the string for a user in a completely different country.

LOWER-LEVEL ISSUES

Higher-level issues, such as the ones just discussed, are likely to affect all applications. But a lot of code that gets written needs to work at a lower level—either accessing memory in strange ways or depending on tricks in assembly language, for example. The remainder of this article will take a look at some of those issues.

LOW MEMORY GLOBALS

Applications should avoid reliance on low-memory globals. In particular, undocumented low-memory globals must be avoided since they are most likely to change. But even dependence on well-known globals can be avoided. For example, the `TickCount` trap returns the same value as the low memory global `Ticks`. `TickCount` is supported under A/UX, while `Ticks` is not, so use of the trap guarantees compatibility. In general, if a trap is available, always use it. And if a glue routine is available, you should use it as well. Then if a change is necessary, you need only update your development system and recompile to implement the change. For the same reason, use of glue routines is also good advice for assembly-language programming.

There is an exception to this rule. The Journaling Driver (see IM I-261) patches key Event Manager traps : `GetMouse`, `Button`, `GetKeys`, and `TickCount`. The Journaling Driver is now used exclusively by MacroMaker™, and unfortunately the driver's patches are not reentrant. This means you cannot safely use these traps in interrupt or VBL code. If you experience strange system hangs only when MacroMaker is installed, this is probably the cause and your code should instead reference the appropriate low-memory globals for the information you need.

SELF-MODIFYING CODE

Applications that use self-modifying code can present serious compatibility issues. There are two kinds of self-modifying code. The first kind involves actually changing machine instructions on the fly. Such code, popular in copy protection schemes, crashes and burns on Macintoshes that use an instruction cache. For example, after a sequence of instructions has been executed and cached by the Macintosh II, some code comes along, modifies the original instructions, and tries to execute them again. But the CPU says, "Ah ha! I already know what these instructions are" and tries to execute the cached instructions, which is not what the programmer originally intended. Fortunately, the Macintosh II and natural selection have made such self-modifying code virtually extinct.

A second, subtler form of self-modifying code keeps variables in the code segment itself. A typical example is the use of `DC.W` or `DC.L` directives to allocate variables in the same segment as the actual code. Such code avoids the earlier problem because it is not actually modifying instructions. The catch is that future operating systems may make 'CODE' segments read-only, and when that code tries to write to its variables, it will fail. Of course, read-only use of such data, such as storing string constants within code segments, is valid. It's fine to do this when no alternative is available. You won't crash in the foreseeable future.

A variety of small tasks, such as VBL tasks and completion routines, run asynchronously on the Macintosh. Because they are executed asynchronously, they cannot be assured that register A5, which by convention points to the application's global variables, is valid when they are called. A common technique used in this case was to store a copy of A5 in with the code so these routines could use the saved value to access global variables.

It's possible to avoid such self-modifying code, as the following MPW sample code illustrates. The trick here is that in creating a VBL task you must pass a record describing the task to the system. When the VBL task is invoked, the system sets up register A0 to point to the start of this record. While the record itself does not contain storage for A5, it's simple to embed the VBL task record into a larger record, or in this case a C struct, that does have room for A5, or anything else you deem important, such as a handle. An inline function called at the start of the VBL task converts A0 into a pointer to the record. Then the task can access anything it needs.

```
#include <Events.h>
#include <OSEvents.h>
#include <OSUtils.h>
#include <Dialogs.h>
#include <Packages.h>
#include <Retrace.h>
#include <Traps.h>

#define  INTERVAL    6
#define rInfoDialog  140
#define rStatTextItem      1

/*
 *  These are globals which will be referenced from our VBL Task
 */
long     gCounter;   /* Counter incremented each time our VBL gets called */

/*
 *  Define a struct to keep track of what we need.  Put theVBLTask into the
 *  struct first because its address will be passed to our VBL task in A0
 */
struct VBLRec {
   VBLTask            theVBLTask;       /* the VBL task itself */
   long  VBLA5;       /* saved CurrentA5 where we can find it */
};
typedef struct VBLRec VBLRec, *VBLRecPtr;

/*
 * GetVBLRec returns the address of the VBLRec associated with our VBL task.
 *  This works because on entry into the VBL task, A0 points to the theVBLTask
 *  field in the VBLRec record, which is the first field in the record and
 *  is the address we return.  Note that this method works whether the VBLRec
 *  is allocated globally, in the heap (as long as the record is locked in
 *  memory) or if it is allocated on the stack as is the case in this example.
 *  In the latter case this is OK as long as the procedure which installed the
 *  task does not exit while the task is running.  This trick allows us to get
 *  to the saved A5, but it could also be used to get to anything we wanted to
 *  store in the record.
  */
```

```
VBLRecPtr GetVBLRec ()
     = 0x2008;             /* MOVE.L  A0,D0 */

/*
 *  DoVBL is called only by StartVBL ()
 */
void DoVBL (VRP)
VBLRecPtr  VRP;
{
     gCounter++;                       /* Show we can set a global */
     VRP->theVBLTask.vblCount = INTERVAL; /* Set ourselves to run again */
}

/*
 *  This is the actual VBL task code.  It uses GetVBLRec to get our VBL record
 *  and properly set up A5.  Having done that, it calls DoVBL to increment a
 *  global counter and sets itself to run again.  Because of the vagaries of
 *  MPW C 3.0 optimization, it calls a separate routine to actually access
 *  global variables.  See Tech Note #208 - "Setting and Restoring A5" for the
 *  reasons for this, as well as for a description of SetA5.
 */
void StartVBL ()

{
     long       curA5;
     VBLRecPtr  recPtr;

     recPtr = GetVBLRec ();           /* First get our record */
     curA5 = SetA5 (recPtr->VBLA5);   /* Get the saved A5 */

                    /* Now we can access globals */
     DoVBL (recPtr);                  /* Call another routine to do actual work */

     (void) SetA5 (curA5); /* Restore old A5 */
}
```

```
/*
 *  InstallVBL creates a dialog just to demonstrate that the global variable
 *  is being updated by the VBL Task.  Before installing the VBL, we store
 *  our A5 in the actual VBL Task record, using SetCurrentA5 described in
 *  Tech Note #208.  We'll run the VBL, showing the counter being incremented,
 *  until the mouse button is clicked.  Then we remove the VBL Task, close the
 *  dialog, and remove the mouse down events to prevent the application from
 *  being inadvertently switched by MultiFinder.
 */
void InstallCVBL ()
{
   VBLRec        theVBLRec;
   DialogPtr     infoDPtr;
   DialogRecord              infoDStorage;
   Str255        numStr;
   OSErr         theErr;
   Handle        theItemHandle;
   short         theItemType;
   Rect          theRect;

   gCounter = 0;            /* Initialize our global counter */
   infoDPtr = GetNewDialog (rInfoDialog, (Ptr) &infoDStorage, (WindowPtr) -1);
   DrawDialog (infoDPtr);
   GetDItem (infoDPtr, rStatTextItem, &theItemType, &theItemHandle, &theRect);

   /*
    *  Store the current value of A5 in the MyA5 field.  For more
    *  information on SetCurrentA5, see Tech Note #208 - "Setting and
    *  Restoring A5".
    */
   theVBLRec.VBLA5 = SetCurrentA5 ();
   /* Set the address of our routine */
   theVBLRec.theVBLTask.vblAddr = (VBLProcPtr) StartVBL;
   theVBLRec.theVBLTask.vblCount = INTERVAL; /* Frequency of task, in ticks */
   theVBLRec.theVBLTask.qType = vType;   /* qElement is a VBL task */
   theVBLRec.theVBLTask.vblPhase = 0;
```

```
    /* Now install the VBL task */
    theErr = VInstall ((QElemPtr)&theVBLRec.theVBLTask);

    if (!theErr) {
          do {
              NumToString (gCounter, numStr);
              SetIText (theItemHandle, numStr);
          } while (!Button ());
          theErr = VRemove ((QElemPtr)&theVBLRec.theVBLTask);
                  /* Remove it when done */
    }

    /* Finish up */
    CloseDialog (infoDPtr);     /* Get rid of our dialog */
    FlushEvents (mDownMask, 0); /* Flush all mouse down events */
  }
```

PRIVILEGED INSTRUCTIONS

Under the current Macintosh operating system, the CPU operates in the supervisor state and applications are allowed to use any and all 680x0 instructions, with the lone exception of the Test And Set (TAS) instruction, which is not supported by the hardware. The A/UX operating system forces applications to run in the user state, and applications that use privileged instructions reserved for the supervisor state will fail. Examples of such instructions are MOVE, ANDI, and EORI instructions with the status register (SR) as either the source or the destination. Typically, these instructions are used to alter the condition code register (CCR), which is the low byte of the SR. Using these instructions with the CCR as the source or destination instead of the SR will accomplish the same thing without causing your application grief. Certain floating point instructions such as FSAVE and FRESTORE are also privileged and should be avoided. As we mentioned, A/UX does not allow the use of privileged instructions and is a good test of compatibility in this case.

DIRECT HARDWARE ACCESS

If you think you need direct access to hardware, let Apple know. It may be acceptable on other personal computers to access hardware directly, but it is decidedly anti-social on the Macintosh and absolutely verboten under operating systems with multi-user protection like A/UX. Beware of schemes for copy protection or performance enhancement that rely on direct hardware access. Macintosh hardware has changed in the past, and it will change in the future. Each new machine may mean yet another revision of your application.

TRAP PATCHING

Trap patching is very useful for overriding or enhancing system trap handling. It is used by the system, for example, to correct errors in the Macintosh ROM. Many applications also use it to provide additional functionality. Because it is very difficult to anticipate all the possible side effects of your patch, maintaining compatibility is difficult, too. Before writing a patch, you should decide if it's absolutely essential. Often the results you need can be achieved without the patch.

Suppose, for example, you decide to patch `ExitToShell`. This may sound like an excellent way for your program to get one last chance at closing files or doing whatever other cleanup is necessary before exiting. Whether `ExitToShell` is called in response to a user's Quit command or because of some fatal error condition, your patch would always have a chance to clean up. But rather than having `ExitToShells` all over your code, you could achieve the same result by calling a single, common exit routine that performed the cleanup and then called `ExitToShell`.

If you absolutely must trap patch, here are some general guidelines. Don't make assumptions about the format of the trap dispatch table. In particular, don't try to read or write entries in the trap dispatch table directly—use `GetTrapAddress` and `SetTrapAddress` instead. If your patch only applies to your application, install it in your application heap. Otherwise, install it in the system heap. Application heap patches will be swapped out by MultiFinder when your application is switched out. Because system heap patches will apply to all applications that use the trap, use them only when absolutely necessary.

You cannot assume that a valid A5 world exists when your patch is invoked. Register A5 points to the base of an application's global variables, and A5 world refers to an application's global address space. MultiFinder maintains different A5 worlds for each running application. Your patch cannot assume when it is called that A5 points to your application's global variables. If it needs access to global variables, you must save a copy of A5 before installing your patch. Then the patch needs to preserve the current value of A5, set the saved value, and restore the original A5 on exit. (See Technical Note #208.) Your patch should avoid use of the Memory Manager if the trap could be invoked at interrupt time or if memory could move during your patch.

Finally, you must not tail patch. In a normal patch, your code completes its task and then invokes the standard trap code to complete the patch. In a tail patch, your code regains control after the standard trap code completes. The problem with this technique is that many of the ROM patches are themselves tail patches, and they rely on knowledge of the caller to accomplish their task. If the ROM patch expects to be called from a ROM address, but is instead called by your patch code, it can become confused. If you JSR to invoke the standard trap code, then you are tail patching. The correct way is to JMP to the starting address of the code.

IN CONCLUSION

It may be useful to know that Apple's implementation of Unix, A/UX, offers a major test for compatibility with System 7.0. A/UX provides a very different environment for Macintosh applications, but applications that follow the compatibility guidelines work without alteration under A/UX. If your application works correctly under A/UX, it stands a very good chance of working correctly under System 7.0.

If you've gotten this far, you are likely to avoid Johnny Appledweeb's fate. You obviously are seriously concerned for your customers and willing to go that extra step to minimize future compatibility problems. It may seem at times that Apple goes out of its way to stretch its own rules, but that is not the case. It is simply impossible to foresee all future hardware and software changes. Incompatibility is unfortunately an ongoing battle. Your part of that battle goes beyond this article and requires you to keep abreast of changes as Apple announces them.