New Technical Notes

Macintosh

Developer Support

Graphics Devices Manager Q&As

Devices

Revised by: Developer Support Center June 1993
Written by: Developer Support Center October 1990

This Technical Note contains a collection of Q&As relating to a specific topic—questions you've sent the Developer Support Center (DSC) along with answers from the DSC engineers. While DSC engineers have checked the Q&A content for accuracy, the Q&A Technical Notes don't have the editing and organization of other Technical Notes. The Q&A function is to get new technical information and updates to you quickly, saving the polish for when the information migrates into reference manuals.

Q&As are now included with Technical Notes to make access to technical updates easier for you. If you have comments or suggestions about Q&A content or distribution, please let us know by sending an AppleLink to DEVFEEDBACK. Apple Partners may send technical questions about Q&A content to DEVSUPPORT for resolution.

New Q&As in this Technical Note:
NewGWorld error code -151

NewGWorld error code -151
Date Written: 1/8/93
Last reviewed: 4/1/93

I received an error code of -151 from NewGWorld when creating a very large off-screen bitmap. Does this mean not enough memory? If so, can I count on it not to change in future versions of the system? It's not listed as one of the possible errors in Inside Macintosh Volume VI.

—

The error -151 is cTempMemErr, "failed to allocate memory for temporary structures," or in other words, there wasn't enough temporary memory available in NewGWorld. NewGWorld returns this error after it receives a MemFullErr from NewTempHandle. (See the Memory Manager chapter of Inside Macintosh Volume VI for more information about temporary memory.)

This was inadvertently left out of Inside Macintosh, but does appear in the MPW interface

files. You can count on this error code in future versions of system software; it was accidently omitted.

GWorld direct access and graphics cards with cached GWorlds
Date Written: 12/1/92
Last reviewed: 3/1/93

Our application draws directly to off-screen GWorlds using our own line-drawing routines and then using CopyBits to get the result to the screen. Since we require direct access to our off-screen GWorld's buffer, is it safe for us to continue to not use the keepLocal flag in our NewGWorld calls? I'm worried that some graphics cards, due to their implementation, assume that the only way that an application will draw to a cached GWorld is by making QuickDraw calls.

—

There's no problem with not using the keepLocal flag. However, keep in mind that you should call GetPixBaseAddr to get the base address of the off-screen GWorld. This ensures that you get the correct base address even if the GWorld is cached. In addition, since the accelerator may be drawing asynchronously, you should make sure there are no operations currently pending. Otherwise, you may get garbage if your drawing conflicts with the asynchronous drawing. You can call QDDone to determine whether pending QuickDraw operations are completed. Both routines are documented in *Inside Macintosh* Volume VI.

Drawing dimmed outline across screens with different depths
Date Written: 12/9/91
Last reviewed: 1/27/92

When the OK button is disabled in the System 7 Standard File dialog box, it's drawn in gray. I was looking for sample code on how to do this in a way that's appropriate for multiple screens at various color depths. For example, how should you draw the outline if you have an OK button in a movable modal dialog box with half the OK button on an 8-bit color screen and the other half on a 1-bit monochrome screen?

—

There are two ways to draw the gray (dimmed) outline across several screens in different depths: one uses MakeRGBPat *(Inside Macintosh* Volume V, page 73), the other uses DeviceLoop *(Inside Macintosh* Volume VI, page 21-23). Look for GrayishOutline.p in the Snippets folder on the *Developer CD Series* disc for a code sample that demonstrates both ways.

Drawing into GWorld after using UpdateGWorld
Date Written: 11/21/91
Last reviewed: 12/11/91

When I resize my real-time animation window in System 6, I call UpdateGWorld with the new size, and after that any drawing into the GWorld has no effect. This same code works

perfectly in System 7. What could cause this?

—

You probably can’t draw anything into your GWorld after using UpdateGWorld to resize it because of the clipping region of your GWorld. In system software versions before 7.0, UpdateGWorld always resizes the GWorld’s clipping region proportional to the amount that the GWorld itself is resized. Unfortunately, NewGWorld initializes the clipping region of the

GWorld to the entire QuickDraw coordinate plane, [T:-32767 L:-32767 B:32767 R:32767]. If UpdateGWorld resizes any of these coordinates so that they fall outside this range, the coordinates wrap around to the other end of the signed integer space, and that makes the clipping region empty. Empty clipping regions stop any drawing from happening.

The change in System 7 is that UpdateGWorld explicitly checks for the clipping region [T:-32767 L:-32767 B:32767 R:32767]. If it finds this, it doesn't resize the clipping region. Otherwise, UpdateGWorld acts the same way that it did before System 7.

One of our mottos is, "Never give QuickDraw a chance to do the wrong thing." In keeping with that, we always explicitly set the clipping region of a GWorld whenever we change the size of the GWorld. So after calling NewGWorld, set its clipping region to be coincident with its portRect. After calling UpdateGWorld to resize the GWorld, set its clipping region to be coincident with its new portRect. That way, you'll always have a known environment and you won't have to worry about the change that was made in System 7—and you'll be less susceptible to bugs in this area in the future.

UpdateGWorld dithering bug workaround
Date Written: 11/14/91
Last reviewed: 12/12/91

UpdateGWorld doesn't seem to respond to the ditherPix flag unless color depth changes. The return flag after changing my color table is 0x10000, indicating that color mapping happened but not dithering. Is this a bug?

—

Yes, this is a bug. UpdateGWorld ignores dithering if no depth change is made. It probably won't be changed in the near future. The workaround is as follows:

1. Create a new pixMap with the new color table.
2. Call CopyBits to transfer your image to the newly created pixMap with dithering from the original GWorld's pixMap.
3. Update the GWorld with the new color table without using ditherPix.
4. Use CopyBits from the newly created pixMap without dithering back to the GWorld.

This will give you the same effect as UpdateGWorld with ditherPix.

Determining if a file is read from CD-ROM or hard disk
Date Written: 8/23/91
Last reviewed: 9/24/91

How can we tell whether a particular Macintosh file is being read from a CD-ROM or a hard disk?

—

You can call the Device Manager routine OpenDriver using the ".AppleCD" string to find a driver reference number of the AppleCD SC drive. If there is more than one AppleCD SC drive hooked up, then additional ".AppleCD" driver reference numbers can be obtained by using the PBControl call with a csCode = 97 (WhoIsThere). This command returns a mask of which

SCSI devices are being serviced by the ".AppleCD" driver (that is, which other drives are AppleCD SCs).

The following code returns the driver reference number for an ".AppleCD" driver instance. The input parameter CDDrive specifies which logical AppleCD SC drive in the SCSI chain to open.

```
#define csWhoIsThere    97

typedef unsigned short    Word;
typedef    unsigned long    Long;

typedef struct WhoIsThereRec {
   ParamBlockHeader
   short        ioRefNum;
   short        csCode;
   struct {
       Byte    fill;
       Byte    SCSIMask;
   } csParam;
} WhoIsThereRec;

pascal    OSErr OpenCD(Byte CDDrive, short *ioRefNum) {

   auto    OSErr            osErr;
   auto    short            ioRefNumTemp,
                           CDDriveCount,
                           SCSIID;
   auto    WhoIsThereRec    *pb;

   pb = (WhoIsThereRec *) NewPtrClear(sizeof (*pb));
   osErr = MemError();
   if (0 != pb && noErr == osErr) {
       osErr = OpenDriver("\p.AppleCD", &ioRefNumTemp);
       if (noErr == osErr) {
           (*pb).ioRefNum        = ioRefNumTemp;
           (*pb).csCode        = csWhoIsThere;
           osErr = PBStatus((ParmBlkPtr)pb, false);
           if (noErr == osErr) {
               CDDriveCount = 0;
               for (SCSIID = 0; SCSIID < 7; ++SCSIID) {
                   if (BitTst(&(*pb).csParam.SCSIMask, 7 - SCSIID)) {
                       ++CDDriveCount;
                       if (CDDrive == CDDriveCount) {
                           *ioRefNum = -(32 + SCSIID) - 1;
                           DisposPtr((Ptr) pb);
                           return noErr;
                       }
                   }
               }
               osErr = paramErr;
           }
       }
       DisposPtr((Ptr) pb);
   }
   return osErr;
}
```

You can modify OpenCD to do exactly what it is you need it to do. Or, you might use it to iterate over the logical CD drive numbers from 0 to 6 until OpenCD returns something other than noErr. If you modify the OpenCD routine, you'll need the "AppleCD SC Developers Guide, Revised Edition" (APDA #A7G0023/A, $25.00) to be successful. Note also that all

unused fields of a parameter block used with the ".AppleCD" driver must be set to zero before calling PBControl.

Iterating over the seven possibilities will result in a table of known ".AppleCD" drive entries—perhaps something like the following structure:

```
struct {
 int count;
 short cdDRefNum[7];
} knownCDDRefNum;
```

With this table in hand, you can match the driver reference number for the volume of any file. In most cases you know which volume the file was on when you opened it. Also, the routines PBGetFCBInfo and PBGetCatInfo both return the volume reference number for a file.

Once you determine the volume reference numbers, either traverse the VCB table or call PBGetVInfo to get the driver reference number. This driver reference number is the driver handling all requests made by the File System for your file!

If that driver reference number for the file is in our "knownCDDRefNum" table, then that file resides on an AppleCD SC drive. This technique works only for Apple or Apple-compatible drivers. An alternate approach to the problem is stepping through the driver table and locating all entries with the name ".AppleCD," noting their driver reference numbers, and then following the procedure outlined above to determine if a file is on a volume owned by that driver.

The csCode method is far clearer than this one, and should be used whenever possible. The bottom line is that there is no guaranteed method of locating a CD-ROM drive due to the lack of a standardized driver model.

X-Ref:
Device Manager chapters of *Inside Macintosh* Volumes II, IV, and V
"Finding a Slot for a Driver" note on latest *Developer CD Series* disc

Accessing a Macintosh driver resource fork at accRun time

Date Written: 6/20/91
Last reviewed: 8/13/91

How do I get Macintosh resources from a driver Init file at accRun time? I can't think of a way other that getting the full path name, which is discouraged.

___

Basically, you need to call PBGetFCBInfo at INIT time to grab the filename, directory ID and volume reference number for the INIT's resource fork. You can then store these in your INIT and pull them out at accRun time. At accRun, just call HOpenResFile to get to your resources. This method is _much_ better than using a full pathname, since this still works in the case where the user re-names the folder containing your INIT.

Here's some sample code that does what you need:

```
OSErr GetCurResLocn(short *saveVRefNum,long *saveDirID,StringPtr saveFName)
{
```

```
   FCBPBRec pb;
   OSErr err;
   short theFile;
   Str255 fName;

   theFile = CurResFile();

   pb.ioFCBIndx = 0;
   pb.ioVRefNum = 0;
   pb.ioRefNum = theFile;
   pb.ioNamePtr = saveFName;

   err = PBGetFCBInfo (&pb,false);

   *saveVRefNum = pb.ioFCBVRefNum;
   *saveDirID = pb.ioFCBParID;

   return err;
}

OSErr SetCurResLocn(short saveVRefNum,long saveDirID,StringPtr saveFName,
                    short *newResFile)
{
   short resRef;
   OSErr err;

   HOpenResFile(saveVRefNum,saveDirID,saveFName,fsRdWrPerm);
   err = ResError();
   if (err!=noErr)
       return err;

   UseResFile(resRef); /* <-- needed in case the res. file was prev. open */

   *newResFile = resRef;

   return ResError();
}

void main()
{
   OSErr err;
   short saveVRefNum;
   long saveDirID;
   Str255 saveFName;
   short newResFile;

   err = GetCurResLocn(&saveVRefNum,&saveDirID,saveFName);
   if (err!=noErr)
       return;

   /* … pass control off to computer here (we're an app, so we fake it) … */

   err = SetCurResLocn(saveVRefNum,saveDirID,saveFName,&newResFile);
   if (err!=noErr)
       DebugStr("\pfailed");
}
```

As you can see, this is an application, so you'll have to do some minor modifications (possibly convert to 680x0). It's pretty straightforward, and the HOpenResFile call is included in MPW glue for MPW 3.0 and is a built-in call for System 7.

DAs in background under System 7.0 lack UnitTable entries
Date Written: 3/14/91
Last reviewed: 6/17/91

Under System 6 my driver, which runs all the time, can send a control call to my open DA (because it too is a driver). Under System 7.0 I get badUnitErr errors (-21) because evidently my DA resides in a different process that is inaccessible to my system-resident driver. How can I get around this?

—

DAs in System 7.0 do not actually have UnitEntries unless they are currently running, so your driver cannot call your DA unless the DA is frontmost. What you might consider instead is having the DA perodically issue a call to the driver, asking if there is anything for it at the moment. If you have an entity that hands data to your resident driver, and the DA then requests the data from the resident driver from time to time, you should have a very robust mechanism, albeit a slightly slower one with greater latency.

Macintosh LC SIntInstall & SLOTIRQ Interrupt Handling
Date Written: 1/4/91
Last reviewed: 1/4/91

How can we get our Macintosh driver to talk with hardware using the SLOTIRQ provided on the bus?

—

To get SLOTIRQ to work correctly you need to use SIntInstall to add a slot interrupt queue element for slot $E. The interrupt service routine pointed to by the slot interrupt queue element must clear the interrupt line before returning. The slot interrupt enable bit in the V8 chip referred to by the Macintosh LC developer note (page 26 in the /SLOTIRQ signal description) is enabled during the boot process, so you don't need to worry about it.

SIntInstall is described in *Inside Macintosh* Volume V on pages V-426 through V-428 and in Designing Cards and Drivers for the Macintosh Family on pages 161 through 163. Designing Cards and Drivers for the Macintosh Family also includes an example of a driver on pages 178 through 202.

JMP or JSR When Calling IODone
Date Written: 12/12/90
Last reviewed: 1/16/91

After an I/O call to a Macintosh slot device driver, shouldn't the IODone routine be called by a JSR instead of a JMP instruction in order for a slot device to return to it with D0 set to an appropriate value depending on whether the interrupt was serviced?

—

You only call IODone when the queued up I/O request has been fully completed. If the interrupt handler is completing an asynchronous call, then you need to call IODone. IODone returns via an RTS, and preserves D0. Therefore, what you should do is have your interrupt service routine set D0 and then jump to IODone (via JIODone*). IODone will do its thing return via an RTS instruction. The sequence would be something like:

0. A program is running
1. The interrupt occurs
2. The Device Manager does a JSR to your Interrupt Service Routine (ISR)
3. The I/O request is complete, so your ISR sets D0 and jumps through JIODone
4. IODone does an RTS, which will be back to the Device Manager
5. The Device Manager does an RTE back to the program
6. The program continues

*The J (in JIODone) stands for Jump, so the ISR pushes JIODone onto the stack, which puts the address of what's in JIODone ($8FC) on the stack. This is followed by an RTS instruction which executes it. So JIODone can be thought of as a vector to IODone.

If you are just handling a hardware interrupt or the I/O is not yet complete, don't call IODone. Do an RTS like the example in Designing Cards and Drivers for the Macintosh Family (Chapter 9 source example, starting with the BeginIH
label).

New info on Macintosh Device Manager calls
Date Written: 12/5/90
Last reviewed: 1/16/91

Are Macintosh Device Manager status calls with csCode=1 calls filtered out? Also, are all high-level Device Manager routines always executed synchronously? If all the calls are synchronous, why is there a high level KillIO routine (to terminate current and pending processes)?

—

Yes, a Status call made with a csCode of 1 never calls your driver. Instead, it returns (in the csParam field) the handle to your driver's Device Control Entry from the Unit Table.

High-level Device Manager calls are executed synchronously. Only the low-level calls can be specified to execute asynchronously. The high-level KillIO routine is useful for terminating I/O pending from a low-level call, which may have been initiated by someone else.

X-Refs: *Inside Macintosh* Volume II, Chapter 6

Macintosh Device Manager handles queuing and asynchronous calls
Date Written: 5/3/89
Last reviewed: 12/17/90

How can the Macintosh Device Manager help my driver handle things like asynchronous

calls, queueing, and so on?

—

The Device Manager will handle all the queueing and asynchronous niceties for you with the jFetch, jStash, and jIODone calls. jIODone handles queueing and calling completion routines for asynchronous calls. jFetch, and jStash are for the interrupt handlers, and provide the mechanism for knowing how far along you are in a particular request.

When you call jIODone, the entry that was being handled will be removed from the queue, its completion routine is called, and then if there is another entry in the queue, your driver will be called to handle it.

If you return via an RTS rather than jIODone (you haven't yet completed the operation requested—that is, waiting for more bytes), the queue entry will remain in the queue, and others behind it will not execute until the next jIODone is called. Queued entries are always executed in sequence. If your driver is not asynchronous, you still need to call jIODOne to clear the queue for the next entry (unless the IMMED bit is set in the ioTrap field).

jFetch and jStash are two calls provided to assist an interrupt driven asynchronous driver. From within your interrupt routine, you can call jFetch to get the next byte from the caller's buffer, or jStash to place the next byte in the caller's buffer. In addition to dealing with the caller's buffer, these two calls also handle the ioActCount and let you know when you have completed the current request (see *Inside Macintosh* Volume II, pages 194-195).

A good example of this is the serial drivers. In the case of the output driver, if the calling application makes a pbWrite call to the serial output driver, and specifies a buffer of 200 characters, the Device Manager places the request in the queue for the Serial Driver and calls the prime routine. The prime routine gets the first character from the buffer by using the jFetch call, and places it in the transmit buffer of the SCC chip and simply does an RTS. The SCC chip has the ability to generate an interrupt when the transmit buffer is empty, so when that happens, the interrupt handler gets the next character from jFetch, and sticks it in the transmit buffer. When jFetch finally gets the last character, the interrupt routine places it in the transmit buffer, and calls jIODone, which then removes that request from the queue, calls the completion routine, if any, and executes the next queued request, if any.

If your driver is supporting non-interrupt driven hardware, you can just receive the prime call, and use jFetch (for example) to get the requested bytes from the buffer, send them to your hardware, and repeat until jFetch returns buffer empty, then return via jIODone. This is an example of synchronous operation that still uses the queueing mechanism, except that no new requests will ever have a chance to get queued until the previous request is completed.

X-Ref:
"Device Manager," *Inside Macintosh* Volumes II and IV

Given a Macintosh gdRefNum, how can I find the associated slot?
Date Written: 3/9/90
Last reviewed: 12/17/90

Given a Macintosh gdRefNum, how can I find the associated slot?

—

Get the slot number from the auxiliary DCE. The following code snippet indexes through the

GDevices (it's assumed the check showed the presence of Color QuickDraw), and pulls the slot number from each GDevice record:

```
gGDHandle := GetDeviceList; {get the first GDevice list handle}
repeat
  if gGDHandle <> nil then
    begin
      gAuxDCEHandle := AuxDCEHandle(GetDCtlEntry(gGDHandle^^.gdRefNum));
```

```
      { do whatever slot specific work is desired, now that the slot }
      { number is known }
      gGDHandle := GetNextDevice(gGDHandle);
      { pass in present GDHandle; the next one is returned }
    end;
  until gGDHandle = nil;
```

X-Refs:
"Device Manager," *Inside Macintosh* Volumes II and IV
"Graphics Devices," *Inside Macintosh* Volume V

Macintosh journaling mechanism

Date Written: 5/3/89
Last reviewed: 12/17/90

How can I use the journaling mechanism described in *Inside Macintosh?*

—

The old journaling mechanism isn't supported any more. It is no longer necessary because of MacroMaker and similar products. MacroMaker now "owns" the older driver, and any future journaling will be done through MacroMaker. Currently there is no technical documentation available for MacroMaker. The current abilities of MacroMaker may not support what some developers will want to do with journaling. Future versions of MacroMaker may add more features.

How do I support locked and ejectable SCSI devices?

Date Written: 5/14/90
Last reviewed: 12/17/90

How do I support locked and ejectable SCSI devices?

—

The only things you should have to support are the _DriveStatus call and modify the DrvQEl record. The rest will be handled by the Macintosh system. The _DriveStatus call gives the information for srvStsCode. This is how the system will know what your disk can support. It is typically only used by floppy (or removable) disk drives. The sample SCSI driver from Apple doesn't need to support it, because it doesn't support any of the information in the _DriveStatus call.

The DrvSts record contains some information of no concern here. The diskInPlace, twoSideFmt, and needsFlush are probably ignored for your device. It's best to zero them out. The DrvSts information is pretty much just the same information returned in the DrvQEl record.

Also, people sometimes get into trouble while developing a driver because current File Manager documentation about the drive queue element is vague. There are 4 bytes in front of a DrvQEl record. These determine the device's abilities, but THESE BYTES ARE NOT ALLOCATED BY THE SYSTEM. When creating the DrvQEl record, you need to add these

four bytes in front of the record yourself. The pointer to a DrvQEl will be the actual record, which is AFTER these 4 bytes. To read these bytes yourself, you'll have to subtract 4 bytes from the DrvQEl pointer.

It is important to note that the Disk Switch error dialog is not an actual dialog, but the system error. It is handled by the same code as SysError which shows the system bomb alert. While this window is present, _SystemTask is not being called. This means the driver will not get an accRun call. To work around this, you will need a VBL task. When the VBL is called it checks the SCSI bus for being free and if so, tests for a new cartridge. Once found, posts a diskInsertEvt. This will be received by the driver.

While the media is inserted, the VBL should not be running until after the cartridge is ejected. Otherwise, the SCSI bus will continue to be accessed unnecessarily, which slows the bus. The VBL task could also slow the system while virtual memory is running.