New Technical Notes

Macintosh

Developer Support

Device Manager Q&As

Devices

Revised by: Developer Support Center May 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 this month:
System 7.0 and SetChooserAlert bug
PBRead/PBWrite instead of FSRead/FSWrite after asynchronous call
Ensuring that Macintosh driver isn't open already

System 7.0 and SetChooserAlert bug

Date Written: 12/28/92
Last reviewed: 3/1/93

With System 7.0, SetChooserAlert doesn't seem to work as documented. It does change the state of bit 6 in HiliteMode, but the "Please change your page setup" message comes up in the Chooser regardless. Is there any workaround for this?

—

The problem you reported is indeed a bug with system software version 7.0 and later. Correcting the system software is the only solution, though this fix may be as simple as a tweak in a future Chooser release (which could come out sooner than a new system software release). Until such a fix is made, SetChooserAlert will only work with system software prior to version 7.0.

PBRead/PBWrite instead of FSRead/FSWrite after asynchronous call

Date Written: 11/24/92

Last reviewed: 3/1/93

I want to send an asynchronous Control command to my driver and while the command is being processed continue work. However, the continued work includes sending FSWrite and FSRead commands to the driver. Is this possible?

—

When you send an asynchronous call to a driver, it gets added to the driver I/O queue. The driver deals with each request in a sequential order. Both FSWrite and FSRead are synchronous. This means that the I/O request is added to the driver queue, and then the Device Manager loops in an internal routine named SyncWait, waiting for ioResult to be less than or equal to 0 (noErr). We assume that you have an interrupt service routine or some other method of completing the I/O request.

If you make a call to either FSWrite or FSRead after doing an asynchronous call, you set up a race condition where the driver is waiting to complete the asynchronous Control call, while the FSRead/Write call is waiting for its completion flag to be marked as done. This won't work.

The solution is to use PBWrite and PBRead instead, and set the asynchronous flag in the call. This way, the driver can properly queue up the calls and handle them in the order issued.

Note that the calls are queued up in a first-in-first-out order. Your control call will complete before your PBRead or PBWrite starts (that is, the Macintosh isn't truly asynchronous in this respect). For instance, if you did the following:

```
PBControl, PBWrite, PBRead, PBRead
```

they'd be executed in precisely that fashion. The last PBRead won't be executed until all three previous calls have finished.

For more information about the way drivers work, you should look on the latest *Developer CD Series* disc for a preliminary copy of the new *Inside Macintosh* chapter on devices. The path to this volume is: Technical Documentation: Inside Macintosh: Devices The new chapter is far better than the old *Inside Macintosh* chapters at explaining how device drivers work. This chapter also has a pretty good explanation of how asynchronous and immediate calls work.

Ensuring that Macintosh driver isn't open already

Date Written: 11/13/92
Last reviewed: 3/1/93

How do I ensure that my driver isn't open already?

—

Check the dOpened flag in your driver's dCtlEntry. This bit is set automatically by the Device Manager when your driver is opened, then cleared when it's closed. However, most drivers should be written so they can take superfluous open calls; many programmers use Open as a method of getting a driver's refNum and opening it at the same time; your driver should check to see if it's already been opened, and if so, do nothing.

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, $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 I*nside 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?

—

Correct. You call IODone when the queued I/O request has been fully completed. If the interrupt handler is completing an asynchronous call, you need to call IODone. IODone returns via an RTS, so the sequence would be something like:

1. Program runs
2. The interrupt occurs
3. Primary interrupt handler code JSRs to secondary (such as VIA) interrupt handler
4. Secondary interrupt handler BSRs to slot handler code
5. Slot handler does a JSR to your Interrupt Service Routine (ISR)
6. The I/O request is complete, so your ISR sets result code in D0 and JSRs to IODone
7. IODone does an RTS, which returns to your ISR
8. Your ISR sets D0 to non-zero and does an RTS to slot handler
9. Slot handler does an RTS back to secondary handler
10. Secondary handler RTSes back to primary handler
11. Primary interrupt handler does an RTE back to the program
12. The program continues

If you're just handling a hardware interrupt or the I/O isn't yet complete, don't call IODone. Do an RTS as in the source code example in Chapter 9 of *Designing Cards and Drivers for the Macintosh Family* (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.