ProDOS 8
#1: The GETLN Buffer and a ProDOS Clock Card

Revised by: Matt Deatherage November 1988
Revised by: Pete McDonald November 1985

This Technical Note describes the effect of a clock card on the GETLN buffer.

ProDOS automatically supports a ThunderClock™ or compatible clock card when the system identifies it as being installed. When programming under ProDOS, always consider the impact of a clock card on the GETLN input buffer ($200 – $2FF). ProDOS can support other clocks which may also use this space.

When ProDOS calls a clock card, the card deposits an ASCII string in the GETLN input buffer in the form: 07,04,14,22,46,57. This string translates as the following:

07 = The month, July (01=Jan,...,12=Dec)
04 = The day of the week, Thurs.(00=Sun,...,06=Sat)
14 = The date (00 to 31)
22 = The hour, 10 p.m. (00 to 23)
46 = The minute (00 to 59)
57 = The second (00 to 59)

ProDOS calls the clock card as part of many of its routines. Anything in the first 17 bytes of the GETLN input buffer is subject to loss if a clock card is installed and is called.

In general, it has been the practice of programmers to use the GETLN input buffer for every conceivable purpose. Therefore, an application should never store anything there. If your application has a future need to know about the contents of the $200 – $2FF space, you should transfer it to some other location to guarantee that it will remain intact, particularly under ProDOS, where a clock card may regularly be overwriting the first 17 bytes.

The *ProDOS 8 Technical Reference Manual* contains more information on the clock driver, including the necessary identification bytes, how the ProDOS driver calls the card, and how you may replace this routine with your own.

Further Reference

- *ProDOS 8 Technical Reference Manual*

ProDOS 8
#2: Porting DOS 3.3 Programs to ProDOS and BASIC.SYSTEM

Revised by: Matt Deatherage November 1988
Revised by: Pete McDonald November 1985

This Technical Note formerly described the `DOSCMD` vector of BASIC.SYSTEM.

This Note formerly described the `DOSCMD` vector of BASIC.SYSTEM, which can be used to let BASIC.SYSTEM interpret ASCII strings as disk commands in much the same way DOS 3.3 did. The *ProDOS 8 Technical Reference Manual* now contains this information in Appendix A.

Further Reference

- *ProDOS 8 Technical Reference Manual*

ProDOS 8
#3: Device Search, Identification, and Driver Conventions

Revised by: Matt Deatherage November 1988
Revised by: Pete McDonald November 1985

This Technical Note formerly described how ProDOS 8 searches for devices and how it deals with devices which are not Disk II drives.

This Note formerly described how ProDOS 8 searches for devices and how it deals with devices which are not Disk II drives; this information is now contained in section 6.3 of the *ProDOS 8 Technical Reference Manual*.

Note: The information on slot mapping on page 113 of the manual and on page 2 of the former edition of this Technical Note is **incorrect**. ProDOS itself will mirror SmartPort devices from slot 5 into slot 2 under certain conditions. Devices should **not** be mirrored into slots other than slot 2. For more information, see ProDOS 8 Technical Note #20, Mirrored Devices and SmartPort.

Further Reference

- *ProDOS 8 Technical Reference Manual*
- ProDOS 8 Technical Note #20, Mirrored Devices and SmartPort

ProDOS 8
#4: I/O Redirection in DOS and ProDOS

Revised by: Matt Deatherage November 1988
Revised by: Pete McDonald November 1985

This Technical Note discusses I/O redirection differences between DOS 3.3 and ProDOS.

Under DOS 3.3, all that is necessary to change the I/O hooks is installing your I/O routine addresses in the character-out vector ($36-$37) and the key-in vector ($38-$39) and notifying DOS (`JSR $3EA`) to take your addresses and swap in its intercept routine addresses.

Under ProDOS, there is no instruction installed at $3EA, so what do you do?

You simply leave the ProDOS BASIC command interpreter's intercept addresses installed at $36-$39 and install your I/O addresses in the global page at $BE30-$BE33. The locations $BE30-$BE31 should contain the output address (normally $FDF0, the Monitor `COUT1` routine), while $BE32-$BE33 should contain the input address (normally $FD1B, the Monitor `KEYIN` routine).

By keeping these vectors in a global page, a special routine for moving the vectors is no longer needed, thus, no $3EA instruction. You install the addresses at their destination yourself.

If you intend to switch between devices (i.e., the screen and the printer), you should save the hooks you find in $BE30-$BE33 and restore them when you are done. Blindly replacing the values in the global page could easily leave you no way to restore input or output to the previous device when you are done.

ProDOS 8
#5: ProDOS Block Device Formatting

Revised by: Matt Deatherage November 1988
Revised by: Pete McDonald October 1985

This Technical Note formerly described the ProDOS `FORMATTER` routine.

The *ProDOS 8 Update Manual* now contains the information about the ProDOS `FORMATTER` routine which this Note formerly described. This routine is available from Apple Software Licensing at Apple Computer, Inc., 20525 Mariani Avenue, M/S 38-I, Cupertino, CA, 95014 or (408) 974-4667.

Note: This routine does not work properly with network volumes on either entry point. You cannot format a network volume with ProDOS 8, nor can you make low-level device calls to it (as `FORMATTER` does in `ENTRY2` to determine the characteristics of a volume). As a general rule, it is better to use `GET_FILE_INFO` to determine the size of the volume since ProDOS MLI calls work with network volumes.

Further Reference

- *ProDOS 8 Update Manual*

ProDOS 8
#6: Attaching External Commands to BASIC.SYSTEM

Revised by: Matt Deatherage November 1988
Revised by: Pete McDonald December 1985

This Technical Note formerly described how to attach an external command to BASIC.SYSTEM.

The *ProDOS 8 Technical Reference Manual*, Appendix A now documents the information which this Note formerly covered about installing an external command into BASIC.SYSTEM to be treated as a normal BASIC.SYSTEM command.

Further Reference

- *ProDOS 8 Technical Reference Manual*

ProDOS 8
#7: Starting and Quitting Interpreter Conventions

Revised by: Matt Deatherage November 1988
Revised by: Pete McDonald December 1985

This Technical Note formerly described conventions for a ProDOS application to start and quit.

Section 5.1.5 of the *ProDOS 8 Technical Reference Manual* now documents the conventions a ProDOS application should follow when starting and quitting, which were formerly covered in this Note as well as ProDOS 8 Technical Note #14, Selector and Dispatcher Conventions.

Further Reference

- *ProDOS 8 Technical Reference Manual*

ProDOS 8
#8: Dealing with /RAM

Revised by: Matt Deatherage November 1988
Written by: Kerry Laidlaw October 1984

This Technical Note formerly described conventions for dealing with the built-in ProDOS 8 RAM disk, /RAM.

Section 5.2.2 of the *ProDOS 8 Technical Reference Manual* now documents the conventions on how to handle /RAM, including how to disconnect it, how to reconnect it, and precautions you should follow if doing either, which were covered in this Note. The manual also includes sample source code.

Executing the sample code which comes with the manual to disconnect /RAM has the undesired effect of decreasing the maximum number of volumes on-line when used with versions of ProDOS 8 prior to 1.2. This side effect is because earlier versions of ProDOS 8 do not have the capability to remove the volume control block (VCB) entry which is allocated for /RAM when it is installed.

In later versions of ProDOS 8 (1.2 and later), this problem no longer exists, and you should issue an `ON_LINE` call to a device after disconnecting it. This call returns error $28 (no device connected), but it also erases the VCB entry for the disconnected device.

Further Reference

- *ProDOS 8 Technical Reference Manual*
- *ProDOS 8 Update Manual*

ProDOS 8
#9: Buffer Management Using BASIC.SYSTEM

Revised by: Matt Deatherage November 1988
Revised by: Pete McDonald October 1985

This Technical Note discusses methods for allocating buffers which will not be arbitrarily deallocated in BASIC.SYSTEM.

Section A.2.1 of the *ProDOS 8 Technical Reference Manual* describes in detail how an application may obtain a buffer from BASIC.SYSTEM for its own use. The buffer will be respected by BASIC.SYSTEM, so if you choose to put a program or other executable code in there, it will be safe.

However, BASIC.SYSTEM does not provide a way to selectively deallocate the buffers it has allocated. Although it is quite easy to allocate space by calling `GETBUFR` ($BEF5) and also quite easy to deallocate by calling `FREEBUFR` ($BEF8), it is not so easy to use `FREEBUFR` to deallocate a particular buffer.

In fact, `FREEBUFR` always deallocates all buffers allocated by `GETBUFR`. This is fine for transient applications, but a method is needed to protect a static code buffer from being deallocated by `FREEBUFR` for a static application.

Location `RSHIMEM` ($BEFB) contains the high byte of the highest available memory location for buffers, normally $96. `FREEBUFR` uses it to determine the beginning page of the highest (or first) buffer. By lowering the value of `RSHIMEM` immediately after the first call to `GETBUFR`, and before any call to `FREEBUFR`, we can fool `FREEBUFR` into not reclaiming all the space. So although it is not possible to selectively deallocate buffers, it is still possible to reserve space that `FREEBUFR` will not reclaim.

Physically, we place the code buffer between BASIC.SYSTEM and its buffers, in the space from $99FF down.

After creating the protected static code buffer, we can call `GETBUFR` and `FREEBUFR` to maintain temporary buffers as needed by our protected module. `FREEBUFR` will not reclaim the protected buffer until after `RSHIMEM` is restored to its original value.

The following is a skeleton example which allocates a two-page buffer for a static code module, protects it from FREEBUFR, then deprotects it and restores it to its original state.

```
START       LDA #$02                ;get 2 pages
            JSR GETBUFR
            LDA RSHIMEM             ;get current RSHIMEM
            SEC                     ;ready for sub
            SBC #$02                ;minus 2 pages
            STA RSHIMEM             ;save new val to fool FREEBUFR
            JSR FREEBUFR            ;CALL FREEBUFR to deallocate.
```

At this point, the value of RSHIMEM is the page number of the beginning of our protected buffer. The static code may now use GETBUFR and FREEBUFR for transient file buffers without fear of freeing its own space from RSHIMEM to $99FF.

To release the protected space, simply restore RSHIMEM to its original value and perform a JSR FREEBUFR.

```
END         LDA RSHIMEM             ;get current val
            CLC                     ;ready for ADD
            ADC #2                  ;give back 2 pages
            STA RSHIMEM             ;tell FREEBUFR about it
            JSR FREEBUFR            ;DO FREEBUFR
            RTS
```

You can reserve any number of pages using this method, as long as the amount you reserve is within available memory limits.

Further Reference

- *ProDOS 8 Technical Reference Manual*

ProDOS 8
#10: Installing Clock Driver Routines

Revised by: Matt Deatherage November 1988
Revised by: Pete McDonald November 1985

This Technical Note formerly described how to install a clock driver routine other than the default.

Section 6.1.1 of the *ProDOS 8 Technical Reference Manual* documents how to install a clock driver other than the default ThunderClock™ driver or the Apple IIGS clock driver into ProDOS 8, which this Note formerly covered.

Further Reference

- *ProDOS 8 Technical Reference Manual*
- ProDOS 8 Technical Note #1, The GETLN Buffer and a ProDOS Clock Card

ProDOS 8
#11: The ProDOS 8 MACHID Byte

Revised by: Matt Deatherage November 1988
Revised by: Pete McDonald November 1985

This Technical Note describes the machine ID byte (MACHID) which ProDOS maintains to help identify different machine types.

ProDOS 8 maintains a machine ID byte, MACHID, at location $BF98 in the ProDOS 8 global page.
Section 5.2.4 of the *ProDOS 8 Technical Reference Manual* correctly documents the definition of this byte.

MACHID has become less robust through the years. Although it can tell you if you are running on an Apple][,][+, IIe, IIc, or Apple /// in emulation mode, it cannot tell you which version of an Apple IIe or IIc you are using, nor can it identify an Apple IIGS (it thinks a IIGS is an Apple IIe). However, the byte still provides a quick test for two components of the system which you might wish to identify: an 80-column card and a clock card.

Bit 1 of MACHID identifies an 80-column card. ProDOS 8 Technical Note #15, How ProDOS 8 Treats Slot 3 explains how this identification is determined. Note that on an Apple IIGS, this bit is always set, even if the user selects Your Card in the Control Panel for slot 3. The bit is set since ProDOS 8 versions 1.7 and later switch out a card in slot 3 in favor of the built-in 80-column firmware, unless the card in slot 3 is an 80-column card. ProDOS 8 behaves in the same manner on an Apple IIe as well.

Bit 0 of MACHID identifies a clock card. Note that on an Apple IIGS, this bit is always set since the IIGS clock cannot be switched out of the system. Due to these unchangeable settings, the value of MACHID on the Apple IIGS is always $B3, as it is on any Apple IIe with an 80-column card and a clock card.

Further Reference

- *ProDOS 8 Technical Reference Manual*
- *Apple IIGS Hardware Reference Manual*
- ProDOS 8 Technical Note #15, How ProDOS 8 Treats Slot 3
- Miscellaneous Technical Note #7, Apple II Family Identification

Apple II
Technical Notes

Developer Technical Support

ProDOS 8
#12: Interrupt Handling

Revised by: Matt Deatherage November 1988
Revised by: Pete McDonald November 1985

This Technical Note clarifies some aspects of ProDOS 8 interrupt handlers.

Although the *ProDOS 8 Technical Reference Manual* (section 6.2) documents interrupt handlers and includes a code example, there still remain a few unclear areas on this subject matter; this Note clarifies these areas.

All interrupt routines must begin with a `CLD` instruction. Although not checked in initial releases of ProDOS 8, this first byte will be checked in future revisions to verify the validity of the interrupt handler.

Although your interrupt handler does not have to disable interrupts (ProDOS 8 does that for you), it must never re-enable interrupts with a 6502 `CLI` instruction. Another interrupt coming through during a non-reentrant interrupt handler can bring the system down.

If your application includes an interrupt handler, you should do the following before exiting:

1. Turn off the interrupt source. Remember, 255 unclaimed interrupts will cause system death.
2. Make a `DEALLOC_INTERRUPT` call before exiting from your application. Do not leave a vector installed that points to a routine that is no longer there.

Within your interrupt handler routines, you must leave all memory banks in the same configuration you found them. Do not forget anything: main language card, main alternate $D000 space, main motherboard ROM, and, on an Apple IIe, IIc, or IIGS, auxiliary language card, auxiliary alternate $D000 space, alternate zero page and stack, etc. This is very important since the ProDOS interrupt receiver assumes that the environment is absolutely unaltered when your handler relinquishes control. In addition, be sure to leave the language card write-enabled.

If your handler recognizes an interrupt and services it, you should clear the carry flag (`CLC`) immediately before returning (`RTS`). If it was not your interrupt, you set set the carry (`SEC`) immediately before returning (`RTS`). Do not use a return from interrupt (`RTI`) to exit; the ProDOS interrupt receiver still has some housekeeping to perform before it issues the `RTI` instruction.

Further Reference

- *ProDOS 8 Technical Reference Manual*

ProDOS 8
#13: Double High-Resolution Graphics Files

Revised by: Matt Deatherage November 1988
Revised by: Pete McDonald November 1985

This Technical Note formerly described a proposed file format for Apple II double high-resolution graphics images.

The information formerly in this Note, the proposed file format for Apple II double high-resolution graphics images, is now covered in the Apple II File Type Notes, File Type $08.

Further Reference

- Apple II File Type Notes, File Type $08

ProDOS 8
#14: Selector and Dispatcher Conventions

Revised by: Matt Deatherage November 1988
Revised by: Pete McDonald December 1985

This Technical Note formerly described conventions for a ProDOS application to start and quit.

Section 5.1.5 of the *ProDOS 8 Technical Reference Manual* now documents the conventions a ProDOS application should follow when starting and quitting, which were formerly covered in this Note as well as ProDOS 8 Technical Note #7, Starting and Quitting Interpreter Conventions.

Further Reference

- *ProDOS 8 Technical Reference Manual*

ProDOS 8
#15: How ProDOS 8 Treats Slot 3

Revised by: Matt Deatherage November 1988
Revised by: Pete McDonald November 1985

This Technical Note describes how ProDOS 8 reacts to non-Apple 80-column cards in slot 3 and how it identifies them.

The *ProDOS 8 Update Manual* now documents much of the information which was originally covered in this Note about how ProDOS 8 reacts to non-Apple 80-column cards in slot 3. However, since there is still some confusion on the issue, we summarize it again in this Note.

On an Apple][+, ProDOS 8 considers the following four Pascal 1.1 protocol ID bytes sufficient to identify a card in slot 3 as an 80-column card and mark the corresponding bit in the `MACHID` byte: $C305 = $38, $C307 = $18, $C30B = $01, $C30C = $8x, where x represents the card's own ID value and is not checked. On any other Apple II, the following fifth ID byte must also match: $C3FA = $2C. This fifth ID byte assures ProDOS 8 that the card supports interrupts on an Apple IIe. Unless ProDOS 8 finds all five ID bytes in an Apple IIe or later, it will not identify the card as an 80-column card and will enable the built-in 80-column firmware instead. In an Apple IIc or IIGS, the internal firmware always matches these five bytes (see below).

If you are designing an 80-column card and wish to meet these requirements, you must follow certain other considerations as well as matching the five identification bytes; the *ProDOS 8 Update Manual* enumerates these other considerations.

The *ProDOS 8 Update Manual* notes that an Apple IIGS does not switch in the 80-column firmware if it is not selected in the Control Panel. However, due to a bug in ProDOS 8 versions 1.6 and earlier, it switches in the 80-column firmware unconditionally. ProDOS 8 cannot respect the Control Panel setting for 80-column firmware in certain situations; it cannot operate in a 128K machine in a 128K configuration (including /RAM) without the presence of the 80-column firmware, since it must utilize the extra 64K. This is just one of the reasons ProDOS 8 does not recognize a card in slot 3 if it is not an 80-column card, as outlined above.

With ProDOS 8 version 1.7 and later, an Apple IIGS behaves exactly like an Apple IIe with respect to slot 3. If a card is slot 3 is selected in the Control Panel, ProDOS 8 ignores it in favor of the built-in 80-column firmware—unless the card matches the five identification bytes listed above. This works the same on a Apple IIe.

Further Reference

- *ProDOS 8 Technical Reference Manual*
- *ProDOS 8 Update Manual*
- ProDOS 8 Technical Note #11, The ProDOS 8 MACHID Byte

Apple II
Technical Notes

Developer Technical Support

ProDOS 8
#16: How to Format a ProDOS Disk Device

Revised by: Matt Deatherage November 1988
Revised by: Pete McDonald November 1985

This Technical Note supplements the *ProDOS 8 Technical Reference Manual* in its description of the low-level driver call that formats the media in a ProDOS device.

The *ProDOS 8 Technical Reference Manual* describes the low-level driver call that formats the media in a ProDOS device, but it neglects to mention the following:

1. It does not work for Disk II drives or /RAM, both of which ProDOS treats specially with built-in driver code.
2. ProDOS has no easy way to tell you whether a device is a Disk II drive or /RAM.

Once ProDOS finishes building its device table, which it does when it boots, it no longer cares about what kind of devices exist, so it does not keep any information about the different types of devices available. ProDOS identifies Disk II devices and installs a built-in driver for them. When it has installed all devices which are physically present, ProDOS then installs /RAM, in a manner similar to Disk II drives, by pointing to the driver code which is within ProDOS itself. This method presents a problem for the developer who wishes to format ProDOS disks since the Disk II driver and the /RAM driver respond to the `FORMAT` request in non-standard ways, yet there is no identification in the global page that tells you which devices are Disk II drives or /RAM.

The Disk II driver does not support the `FORMAT` request, and the /RAM driver responds by "formatting" the RAM disk and also writing to it a virgin directory and bitmap; neither of these two cases is documented in the *ProDOS 8 Technical Reference Manual*. To write special-case code for these two device types, you must be able to identify them, and the method for identification is available in ProDOS 8 Technical Note #21: Identifying ProDOS Devices.

You should note, however, that AppleTalk network volumes cannot be formatted; they return a `DEVICE NOT CONNECTED` error for the `FORMAT` and any low-level device call. You may access AppleTalk network volumes through ProDOS MLI calls only.

Also note that Apple licences a ProDOS 8 Formatter routine, which correctly identifies and handles Disk II drives and /RAM. You should contact Apple Software Licensing at Apple Computer, Inc., 20525 Mariani Avenue, M/S 38-I, Cupertino, CA, 95014 or (408) 974-4667 if you wish to license this routine.

Further Reference

- *ProDOS 8 Technical Reference Manual*
- *ProDOS 8 Update Manual*
- ProDOS 8 Technical Note #21, Identifying ProDOS Devices

Apple II
Technical Notes

Developer Technical Support

ProDOS 8
#17: Recursive ProDOS Catalog Routine

Revised by: Dave Lyons, Keith Rollin, & Matt Deatherage November 1989
Written by: Greg Seitz December 1983

This Technical Note presents an assembly language example of a recursive directory reading routine which is AppleShare compatible.
Changes since November 1988: The routine now ignores the `file_count` field in a directory, and it properly increments `ThisBlock`. More discussion of AppleShare volumes is included.

This Note presents a routine in assembly language for recursively cataloging a ProDOS directory. If you apply this technique to the volume directory of a disk, it will display the name of every file stored on the disk. The routine displays the contents of a given directory (the volume directory in this case), displays the contents of each subdirectory as it is encountered.

`READ_BLOCK` is not used, since it does not work with AppleShare servers. READ is used instead, since it works for AppleShare volumes as well as local disks. Instead of using directory pointers to decide which block to read next, we simply read the directory and display filenames as we go, until we reach a subdirectory file. When we reach a subdirectory, the routine saves our place, plunges down one level of the tree structure, and catalogs the subdirectory. You repeat the process if you find a subdirectory at the current level. When you reach the EOF of any directory, the routine closes the current directory and pops back up one level, and when it reaches the EOF of the initial directory, the routine is finished.

This routine is generally compatible with AppleShare volumes, but it is impossible to guarantee a complete traversal of all the accessible files on an AppleShare volume: another user on the same volume can add or remove files or directories at any time. If entries are added or removed, some filenames may be displayed twice or missed completely. Be sure that your programs deal with this sort of situation adequately.

We assume that AppleShare is in short naming mode (as it is by default under ProDOS 8). If you enable long naming mode, then illegal characters in filenames will not be translated into question marks. In this case, the code would need to be modified to deal with non-ASCII characters. Also, the `ChopName` routine would need to be aware that a slash (/) character could be contained inside the name of a directory that had been added to the pathname. (As the code stands, such directories fail to open, but their names are still temporarily added to the pathname.)

When the catalog routine encounters an error, it displays a brief message and continues. It is important not to abort on an error, since AppleShare volumes generally contain files and folders with names that are inaccessible to ProDOS, as well as folders that are inaccessible to your program's user (error $4E, access error).

The code example includes a simple test of the `ReadDir` routine, which is the actual recursive catalog routine. Note that the simple test relies upon the `GETBUFR` routine in BASIC.SYSTEM to allocate a buffer; therefore, as presented, the routine requires the presence of BASIC.SYSTEM. The actual `ReadDir` routine requires nothing outside of the ProDOS 8 MLI.

```
----- NEXT OBJECT FILE NAME IS CATALOG.0
0800:        0800    2           org   $800
0800:                3 ************************************************
0800:                4 *
0800:                5 * Recursive ProDOS Catalog Routine
0800:                6 *
0800:                7 * by: Greg Seitz 12/83
0800:                8 *     Pete McDonald 1/86
0800:                9 *     Keith Rollin 7/88
0800:               10 *     Dave Lyons 11/89
0800:               11 *
0800:               12 * This program shows the latest "Apple Approved"
0800:               13 * method for reading a directory under ProDOS 8.
0800:               14 * READ_BLOCK is not used, since it is incompatible
0800:               15 * with AppleShare file servers.
0800:               16 *
0800:               17 * November 1989: The file_count field is no longer
0800:               18 * used (all references to ThisEntry were removed).
0800:               19 * This is because the file count can change on the fly
0800:               20 * on AppleShare volumes.  (Note that the old code was
0800:               21 * accidentally decrementing the file count when it
0800:               22 * found an entry for a deleted file, so some files
0800:               23 * could be left off the end of the list.)
0800:               24 *
0800:               25 * Also, ThisBlock now gets incremented when a chunk
0800:               26 * of data is read from a directory.  Previously, this
0800:               27 * routine could get stuck in an endless loop when
0800:               28 * a subdirectory was found outside the first block of
0800:               29 * its parent directory.
0800:               30 *
0800:               31 * Limitations:  This routine cannot reach any
0800:               32 * subdirectory whose pathname is longer than 64
0800:               33 * characters, and it will not operate correctly if
0800:               34 * any subdirectory is more than 255 blocks long
0800:               35 * (because ThisBlock is only one byte).
0800:               36 *
0800:               37 ************************************************
0800:               38 *
0800:               39 * Equates
0800:               40 *
0800:               41 * Zero page locations
0800:               42 *
0800:        0080   43 dirName   equ   $80          ; pointer to directory name
0800:        0082   44 entPtr    equ   $82          ; ptr to current entry
0800:               45 *
0800:               46 * ProDOS command numbers
0800:               47 *
0800:        BF00   48 MLI       equ   $BF00        ; MLI entry point
0800:        00C7   49 mliGetPfx equ   $C7          ; GET_PREFIX
0800:        00C8   50 mliOpen   equ   $C8          ; Open a file command
0800:        00CA   51 mliRead   equ   $CA          ; Read a file command
0800:        00CC   52 mliClose  equ   $CC          ; Close a file command
```

```
0800:        00CE   53 mliSetMark equ  $CE          ; SET_MARK command
0800:        004C   54 EndOfFile equ   $4C          ; EndOfFile error
0800:               55 *
0800:               56 * BASIC.SYSTEM stuff
0800:               57 *
0800:        BEF5   58 GetBufr   equ   $BEF5        ; BASIC.SYSTEM get buffer routine
```

```
01 CATALOG          ProDOS Catalog Routine              14-OCT-89  16:20 PAGE 3

0800:                59 *
0800:                60 * Offsets into the directory
0800:                61 *
0800:        0000    62 oType     equ   $0            ; offset to file type byte
0800:        0023    63 oEntLen   equ   $23           ; length of each dir. entry
0800:        0024    64 oEntBlk   equ   $24           ; entries in each block
0800:                65 *
0800:                66 * Monitor routines
0800:                67 *
0800:        FDED    68 cout      equ   $FDED         ; output a character
0800:        FD8E    69 crout     equ   $FD8E         ; output a RETURN
0800:        FDDA    70 prbyte    equ   $FDDA         ; print byte in hex
0800:        00A0    71 space     equ   $A0           ; a space character
0800:                72 *
0800:                73 ******************************************************
0800:                74 *
0800:        0800    75 Start     equ   *
0800:                76 *
0800:                77 * Simple routine to test the recursive ReadDir
0800:                78 * routine. It gets an I/O buffer for ReadDir, gets
0800:                79 * the current prefix, sets the depth of recursion
0800:                80 * to zero, and calls ReadDir to process all of the
0800:                81 * entries in the directory.
0800:                82 *
0800:A9 04           83           lda   #4            ; get an I/O buffer
0802:20 F5 BE        84           jsr   GetBufr
0805:B0 17   081E    85           bcs   exit          ; didn't get it
0807:8D D7 09        86           sta   ioBuf+1
080A:                87 *
080A:                88 * Use the current prefix for the name of the
080A:                89 * directory to display.  Note that the string we
080A:                90 * pass to ReadDir has to end with a "/", and that
080A:                91 * the result of GET_PREFIX does.
080A:                92 *
080A:20 00 BF        93           jsr   MLI
080D:C7              94           db    mliGetPfx
080E:E8 09           95           dw    GetPParms
0810:B0 0C   081E    96           bcs   exit
0812:                97 *
0812:A9 00           98           lda   #0
0814:8D CE 09        99           sta   Depth
0817:               100 *
0817:A9 EB          101           lda   #nameBuffer
0819:A2 0B          102           ldx   #<nameBuffer
081B:20 1F 08       103           jsr   ReadDir
081E:               104 *
081E:        081E   105 exit      equ   *
081E:60             106           rts
081F:               107 *
081F:               108 ******************************************************
081F:               109 ******************************************************
081F:               110 *
081F:        081F   111 ReadDir   equ   *
081F:               112 *
081F:               113 *  This is the actual recursive routine. It takes as
081F:               114 *  input a pointer to the directory name to read in
081F:               115 *  A,X (lo,hi), opens it, and starts to read the
081F:               116 *  entries. When it encounters a filename, it calls
```

```
01 CATALOG         ProDOS Catalog Routine              14-OCT-89  16:20 PAGE 4

081F:              117 *  the routine "VisitFile". When it encounters a
081F:              118 *  directory name, it calls "VisitDir".
081F:              119 *
081F:              120 *  The directory pathname string must end with a "/"
081F:              121 *  character.
081F:              122 *
081F:              123 ******************************************************
081F:              124 *
081F:85 80         125           sta   dirName       ; save a pointer to name
0821:86 81         126           stx   dirName+1
0823:              127 *
0823:8D D4 09      128           sta   openName      ; set up OpenFile params
0826:8E D5 09      129           stx   openName+1
0829:              130 *
0829:        0829  131 ReadDir1  equ   *             ; recursive entry point
0829:20 79 08      132           jsr   OpenDir       ; open the directory as a file
082C:B0 1F   084D  133           bcs   done
082E:              134 *
082E:4C 48 08      135           jmp   nextEntry     ; jump to the end of the loop
0831:              136 *
0831:        0831  137 loop      equ   *
0831:A0 00         138           ldy   #oType        ; get type of current entry
0833:B1 82         139           lda   (entPtr),y
0835:29 F0         140           and   #$F0          ; look at 4 high bits
0837:C9 00         141           cmp   #0            ; inactive entry?
0839:F0 0D   0848  142           beq   nextEntry     ; yes - bump to next one
083B:C9 D0         143           cmp   #$D0          ; is it a directory?
083D:F0 06   0845  144           beq   ItsADir       ; yes, so call VisitDir
083F:20 B3 08      145           jsr   VisitFile     ; no, it's a file
0842:4C 48 08      146           jmp   nextEntry
0845:              147 *
0845:20 BA 08      148 ItsADir   jsr   VisitDir
0848:        0848  149 nextEntry equ   *
0848:20 77 09      150           jsr   GetNext       ; get pointer to next entry
084B:90 E4   0831  151           bcc   loop          ; Carry set means we're done
084D:        084D  152 done      equ   *             ; moved before PHA (11/89 DAL)
084D:48            153           pha                 ; save error code
084E:              154 *
084E:20 00 BF      155           jsr   MLI           ; close the directory
0851:CC            156           db    mliClose
0852:E1 09         157           dw    CloseParms
0854:              158 *
0854:68            159           pla                 ;we're expecting EndOfFile error
0855:C9 4C         160           cmp   #EndOfFile
0857:F0 1F   0878  161           beq   hitDirEnd
0859:              162 *
0859:              163 * We got an error other than EndOfFile--report the
0859:              164 * error clumsily ("ERR=$xx").
0859:              165 *
0859:48            166           pha
085A:A9 C5         167           lda   #'E'|$80
085C:20 ED FD      168           jsr   cout
085F:A9 D2         169           lda   #'R'|$80
0861:20 ED FD      170           jsr   cout
0864:20 ED FD      171           jsr   cout
0867:A9 BD         172           lda   #'='|$80
0869:20 ED FD      173           jsr   cout
086C:A9 A4         174           lda   #'$'|$80
```

```
01 CATALOG         ProDOS Catalog Routine              14-OCT-89  16:20 PAGE 5

086E:20 ED FD      175           jsr   cout
0871:68            176           pla
0872:20 DA FD      177           jsr   prbyte
0875:20 8E FD      178           jsr   crout
0878:              179 *
0878:        0878  180 hitDirEnd equ   *
0878:60            181           rts
0879:              182 *
0879:              183 ****************************************************
0879:              184 *
0879:        0879  185 OpenDir   equ   *
0879:              186 *
0879:              187 *  Opens the directory pointed to by OpenParms
0879:              188 *  parameter block. This pointer should be init-
0879:              189 *  ialized BEFORE this routine is called. If the
0879:              190 *  file is successfully opened, the following
0879:              191 *  variables are set:
0879:              192 *
0879:              193 *     xRefNum     ; all the refnums
0879:              194 *     entryLen    ; size of directory entries
0879:              195 *     entPtr     ; pointer to current entry
0879:              196 *     ThisBEntry  ; entry number within this block
0879:              197 *     ThisBlock   ; offset (in blocks) into dir.
0879:              198 *
0879:20 00 BF      199           jsr   MLI           ; open dir as a file
087C:C8            200           db    mliOpen
087D:D3 09         201           dw    OpenParms
087F:B0 31   08B2  202           bcs   OpenDone
0881:              203 *
0881:AD D8 09      204           lda   oRefNum       ; copy the refnum return-
0884:8D DA 09      205           sta   rRefNum       ; ed by Open into the
0887:8D E2 09      206           sta   cRefNum       ; other param blocks.
088A:8D E4 09      207           sta   sRefNum
088D:              208 *
088D:20 00 BF      209           jsr   MLI           ; read the first block
0890:CA            210           db    mliRead
0891:D9 09         211           dw    ReadParms
0893:B0 1D   08B2  212           bcs   OpenDone
0895:              213 *
0895:AD 0E 0A      214           lda   buffer+oEntLen ; init 'entryLen'
0898:8D D1 09      215           sta   entryLen
089B:              216 *
089B:A9 EF         217           lda   #buffer+4     ; init ptr to first entry
089D:85 82         218           sta   entPtr
089F:A9 09         219           lda   #<buffer+4
08A1:85 83         220           sta   entPtr+1
08A3:              221 *
08A3:AD 0F 0A      222           lda   buffer+oEntblk ; init these values based on
08A6:8D CF 09      223           sta   ThisBEntry    ; values in the dir header
08A9:8D D2 09      224           sta   entPerBlk
08AC:              225 *
08AC:A9 00         226           lda   #0            ; init block offset into dir.
08AE:8D D0 09      227           sta   ThisBlock
08B1:              228 *
08B1:18            229           clc                 ; say that open was OK
08B2:              230 *
08B2:        08B2  231 OpenDone  equ   *
08B2:60            232           rts
```

```
01 CATALOG         ProDOS Catalog Routine              14-OCT-89  16:20 PAGE 6

08B3:              233 *
08B3:              234 ***************************************************
08B3:              235 *
08B3:        08B3  236 VisitFile equ   *
08B3:              237 *
08B3:              238 * Do whatever is necessary when we encounter a
08B3:              239 * file entry in the directory. In this case, we
08B3:              240 * print the name of the file.
08B3:              241 *
08B3:20 AC 09      242           jsr   PrintEntry
08B6:20 8E FD      243           jsr   crout
08B9:60            244           rts
08BA:              245 *
08BA:              246 ***************************************************
08BA:              247 *
08BA:        08BA  248 VisitDir  equ   *
08BA:              249 *
08BA:              250 * Print the name of the subdirectory we are looking
08BA:              251 * at, appending a "/" to it (to indicate that it's
08BA:              252 * a directory), and then calling RecursDir to list
08BA:              253 * everything in that directory.
08BA:              254 *
08BA:20 AC 09      255           jsr   PrintEntry    ; print dir's name
08BD:A9 AF         256           lda   #'/'|$80      ; tack on / at end
08BF:20 ED FD      257           jsr   cout
08C2:20 8E FD      258           jsr   crout
08C5:              259 *
08C5:20 C9 08       260            jsr   RecursDir     ; enumerate all entries in sub-
dir.
08C8:              261 *
08C8:60            262           rts
08C9:              263 *
08C9:              264 ***************************************************
08C9:              265 *
08C9:        08C9  266 RecursDir equ   *
08C9:              267 *
08C9:              268 * This routine calls ReadDir recursively. It
08C9:              269 *
08C9:              270 * - increments the recursion depth counter,
08C9:              271 * - saves certain variables onto the stack
08C9:              272 * - closes the current directory
08C9:              273 * - creates the name of the new directory
08C9:              274 * - calls ReadDir (recursively)
08C9:              275 * - restores the variables from the stack
08C9:              276 * - restores directory name to original value
08C9:              277 * - re-opens the old directory
08C9:              278 * - moves to our last position within it
08C9:              279 * - decrements the recursion depth counter
08C9:              280 *
08C9:EE CE 09      281           inc   Depth         ; bump this for recursive call
08CC:              282 *
08CC:              283 * Save everything we can think of (the women,
08CC:              284 * the children, the beer, etc.).
08CC:              285 *
08CC:A5 83         286           lda   entPtr+1
08CE:48            287           pha
08CF:A5 82         288           lda   entPtr
08D1:48            289           pha
08D2:AD CF 09      290           lda   ThisBEntry
```

```
01 CATALOG          ProDOS Catalog Routine              14-OCT-89  16:20 PAGE 7

08D5:48             291           pha
08D6:AD D0 09       292           lda   ThisBlock
08D9:48             293           pha
08DA:AD D1 09       294           lda   entryLen
08DD:48             295           pha
08DE:AD D2 09       296           lda   entPerblk
08E1:48             297           pha
08E2:               298 *
08E2:               299 * Close the current directory, as ReadDir will
08E2:               300 * open files of its own, and we don't want to
08E2:               301 * have a bunch of open files lying around.
08E2:               302 *
08E2:20 00 BF       303           jsr   MLI
08E5:CC             304           db    mliClose
08E6:E1 09          305           dw    CloseParms
08E8:               306 *
08E8:20 2F 09       307           jsr   ExtendName    ; make new dir name
08EB:               308 *
08EB:20 29 08       309           jsr   ReadDir1      ; enumerate the subdirectory
08EE:               310 *
08EE:20 65 09       311           jsr   ChopName      ; restore old directory name
08F1:               312 *
08F1:20 79 08       313           jsr   OpenDir       ; re-open it back up
08F4:90 01   08F7   314           bcc   reOpened
08F6:               315 *
08F6:               316 * Can't continue from this point--exit in
08F6:               317 * whatever way is appropriate for your
08F6:               318 * program.
08F6:               319 *
08F6:00             320           brk
08F7:               321 *
08F7:        08F7   322 reOpened  equ   *
08F7:               323 *
08F7:               324 * Restore everything that we saved before
08F7:               325 *
08F7:68             326           pla
08F8:8D D2 09       327           sta   entPerBlk
08FB:68             328           pla
08FC:8D D1 09       329           sta   entryLen
08FF:68             330           pla
0900:8D D0 09       331           sta   ThisBlock
0903:68             332           pla
0904:8D CF 09       333           sta   ThisBEntry
0907:68             334           pla
0908:85 82          335           sta   entPtr
090A:68             336           pla
090B:85 83          337           sta   entPtr+1
090D:               338 *
090D:A9 00          339           lda   #0
090F:8D E5 09       340           sta   Mark
0912:8D E7 09       341           sta   Mark+2
0915:AD D0 09       342           lda   ThisBlock     ; reset last position in dir
0918:0A             343           asl   a             ; = to block # times 512
0919:8D E6 09       344           sta   Mark+1
091C:2E E7 09       345           rol   Mark+2
091F:               346 *
091F:20 00 BF       347           jsr   MLI           ; reset the file marker
0922:CE             348           db    mliSetMark
```

```
01 CATALOG          ProDOS Catalog Routine              14-OCT-89  16:20 PAGE 8

0923:E3 09          349           dw    SetMParms
0925:               350 *
0925:20 00 BF       351           jsr   MLI            ; now read in the block we
0928:CA             352           db    mliRead        ; were on last.
0929:D9 09          353           dw    ReadParms
092B:               354 *
092B:CE CE 09       355           dec   Depth
092E:60             356           rts
092F:               357 *
092F:               358 ********************************************************
092F:               359 *
092F:        092F   360 ExtendName equ  *
092F:               361 *
092F:               362 * Append the name in the current directory entry
092F:               363 * to the name in the directory name buffer. This
092F:               364 * will allow us to descend another level into the
092F:               365 * disk hierarchy when we call ReadDir.
092F:               366 *
092F:A0 00          367           ldy   #0             ; get length of string to copy
0931:B1 82          368           lda   (entPtr),y
0933:29 0F          369           and   #$0F
0935:8D 62 09       370           sta   extCnt         ; save the length here
0938:8C 63 09       371           sty   srcPtr         ; init src ptr to zero
093B:               372 *
093B:A0 00          373           ldy   #0             ; init dest ptr to end of
093D:B1 80          374           lda   (dirName),y    ; the current directory name
093F:8D 64 09       375           sta   destPtr
0942:               376 *
0942:        0942   377 extloop   equ   *
0942:EE 63 09       378           inc   srcPtr         ; bump to next char to read
0945:EE 64 09       379           inc   destPtr        ; bump to next empty location
0948:AC 63 09       380           ldy   srcPtr         ; get char of sub-dir name
094B:B1 82          381           lda   (entPtr),y
094D:AC 64 09       382           ldy   destPtr        ; tack on to end of cur. dir.
0950:91 80          383           sta   (dirName),y
0952:CE 62 09       384           dec   extCnt         ; done all chars?
0955:D0 EB   0942   385           bne   extloop        ; no - so do more
0957:               386 *
0957:C8             387           iny
0958:A9 2F          388           lda   #'/'           ; tack "/" on to the end
095A:91 80          389           sta   (dirName),y
095C:               390 *
095C:98             391           tya                  ; fix length of filename to open
095D:A0 00          392           ldy   #0
095F:91 80          393           sta   (dirName),y
0961:               394 *
0961:60             395           rts
0962:               396 *
0962:        0001   397 extCnt    ds    1
0963:        0001   398 srcPtr    ds    1
0964:        0001   399 destPtr   ds    1
0965:               400 *
0965:               401 *
0965:               402 ********************************************************
0965:               403 *
0965:        0965   404 ChopName  equ   *
0965:               405 *
0965:               406 * Scans the current directory name, and chops
```

```
01 CATALOG         ProDOS Catalog Routine              14-OCT-89  16:20 PAGE 9

0965:              407 * off characters until it gets to a /.
0965:              408 *
0965:A0 00         409           ldy   #0            ; get len of current dir.
0967:B1 80         410           lda   (dirName),y
0969:A8            411           tay
096A:        096A  412 ChopLoop  equ   *
096A:88            413           dey                 ; bump to previous char
096B:B1 80         414           lda   (dirName),y
096D:C9 2F         415           cmp   #'/'
096F:D0 F9   096A  416           bne   ChopLoop
0971:98            417           tya
0972:A0 00         418           ldy   #0
0974:91 80         419           sta   (dirName),y
0976:60            420           rts
0977:              421 *
0977:              422 ****************************************************
0977:              423 *
0977:        0977  424 GetNext   equ   *
0977:              425 *
0977:              426 * This routine is responsible for making a pointer
0977:              427 * to the next entry in the directory. If there are
0977:              428 * still entries to be processed in this block, then
0977:              429 * we simply bump the pointer by the size of the
0977:              430 * directory entry. If we have finished with this
0977:              431 * block, then we read in the next block, point to
0977:              432 * the first entry, and increment our block counter.
0977:              433 *
0977:CE CF 09       434           dec   ThisBEntry    ; dec count for this block
097A:F0 10   098C  435           beq   ReadNext      ; done w/this block, get next one
097C:              436 *
097C:18            437           clc                 ; else bump up index
097D:A5 82         438           lda   entPtr
097F:6D D1 09      439           adc   entryLen
0982:85 82         440           sta   entPtr
0984:A5 83         441           lda   entPtr+1
0986:69 00         442           adc   #0
0988:85 83         443           sta   entPtr+1
098A:18            444           clc                 ; say that the buffer's good
098B:60            445           rts
098C:              446 *
098C:        098C  447 ReadNext  equ   *
098C:20 00 BF      448           jsr   MLI           ; get the next block
098F:CA            449           db    mliRead
0990:D9 09         450           dw    ReadParms
0992:B0 16   09AA  451           bcs   DirDone
0994:              452 *
0994:EE D0 09      453           inc   ThisBlock
0997:              454 *
0997:A9 EF         455           lda   #buffer+4     ; set entry pointer to beginning
0999:85 82         456           sta   entPtr        ; of first entry in block
099B:A9 09         457           lda   #<buffer+4
099D:85 83         458           sta   entPtr+1
099F:              459 *
099F:AD D2 09      460           lda   entPerBlk     ; re-init 'entries in this block'
09A2:8D CF 09      461           sta   ThisBEntry
09A5:CE CF 09      462           dec   ThisBEntry
09A8:18            463           clc                 ; return 'No error'
09A9:60            464           rts
```

```
01 CATALOG          ProDOS Catalog Routine              14-OCT-89  16:20 PAGE 10

09AA:               465 *
09AA:         09AA  466 DirDone   equ   *
09AA:38             467           sec               ; return 'an error occurred' (error
in A)
09AB:60             468           rts
09AC:               469 *
09AC:               470 ********************************************************
09AC:               471 *
09AC:         09AC  472 PrintEntry equ  *
09AC:               473 *
09AC:               474 * Using the pointer to the current entry, this
09AC:               475 * routine prints the entry name. It also pays
09AC:               476 * attention to the recursion depth, and indents
09AC:               477 * by 2 spaces for every level.
09AC:               478 *
09AC:AD CE 09        479             lda   Depth           ; indent two blanks for each
level
09AF:0A             480           asl   a             ; of directory nesting
09B0:AA             481           tax
09B1:F0 08   09BB  482           beq   spcDone
09B3:A9 A0          483 spcloop   lda   #space
09B5:20 ED FD       484           jsr   cout
09B8:CA             485           dex
09B9:D0 F8   09B3  486           bne   spcloop
09BB:         09BB  487 spcDone   equ   *
09BB:               488 *
09BB:A0 00           489            ldy   #0             ; get byte that has the length
byte
09BD:B1 82          490           lda   (entPtr),y
09BF:29 0F          491           and   #$0F          ; get just the length
09C1:AA             492           tax
09C2:         09C2  493 PrntLoop  equ   *
09C2:C8             494           iny                 ; bump to the next char.
09C3:B1 82          495           lda   (entPtr),y    ; get next char
09C5:09 80          496           ora   #$80          ; COUT likes high bit set
09C7:20 ED FD       497           jsr   cout          ; print it
09CA:CA             498           dex                 ; printed all chars?
09CB:D0 F5   09C2  499           bne   PrntLoop      ; no - keep going
09CD:60             500           rts
09CE:               501 *
09CE:               502 ********************************************************
09CE:               503 *
09CE:               504 * Some global variables
09CE:               505 *
09CE:         0001  506 Depth     ds    1             ; amount of recursion
09CF:         0001  507 ThisBEntry ds   1             ; entry in this block
09D0:         0001  508 ThisBlock ds    1             ; block with dir
09D1:         0001  509 entryLen  ds    1             ; length of each directory entry
09D2:         0001  510 entPerBlk ds    1             ; entries per block
09D3:               511 *
09D3:               512 ********************************************************
09D3:               513 *
09D3:               514 * ProDOS command parameter blocks
09D3:               515 *
09D3:         09D3  516 OpenParms equ   *
09D3:03             517           db    3             ; number of parms
09D4:         0002  518 OpenName  ds    2             ; pointer to filename
09D6:00 00          519 ioBuf     dw    $0000         ; I/O buffer
09D8:         0001  520 oRefNum   ds    1             ; returned refnum
09D9:               521 *
09D9:         09D9  522 ReadParms equ   *
```

```
01 CATALOG          ProDOS Catalog Routine              14-OCT-89  16:20 PAGE 11

09D9:04             523           db    4              ; number of parms
09DA:         0001  524 rRefNum   ds    1              ; refnum from Open
09DB:EB 09          525           dw    buffer         ; pointer to buffer
09DD:00 02          526 reqAmt    dw    512            ; amount to read
09DF:         0002  527 retAmt    ds    2              ; amount actually read
09E1:               528 *
09E1:         09E1  529 CloseParms equ  *
09E1:01             530           db    1              ; number of parms
09E2:         0001  531 cRefNum   ds    1              ; refnum from Open
09E3:               532 *
09E3:         09E3  533 SetMParms equ   *
09E3:02             534           db    2              ; number of parms
09E4:         0001  535 sRefNum   ds    1              ; refnum from Open
09E5:         0003  536 Mark      ds    3              ; file position
09E8:               537 *
09E8:         09E8  538 GetPParms equ   *
09E8:01             539           db    1              ; number of parms
09E9:EB 0B          540           dw    nameBuffer     ; pointer to buffer
09EB:               541 *
09EB:         0200  542 buffer    ds    512            ; enough for whole block
0BEB:               543 *
0BEB:         0040  544 nameBuffer ds   64             ; space for directory name

01 SYMBOL TABLE     SORTED BY SYMBOL                    14-OCT-89  16:20 PAGE 12

 09EB BUFFER          096A CHOPLOOP        0965 CHOPNAME        09E1 CLOSEPARMS
 FDED COUT            09E2 CREFNUM         FD8E CROUT           09CE DEPTH
 0964 DESTPTR         09AA DIRDONE           80 DIRNAME         084D DONE
   4C ENDOFFILE       09D2 ENTPERBLK         82 ENTPTR          09D1 ENTRYLEN
 081E EXIT            0962 EXTCNT          092F EXTENDNAME      0942 EXTLOOP
 BEF5 GETBUFR         0977 GETNEXT         09E8 GETPPARMS       0878 HITDIREND
 09D6 IOBUF           0845 ITSADIR         0831 LOOP            09E5 MARK
   CC MLICLOSE          C7 MLIGETPFX         C8 MLIOPEN         BF00 MLI
   CA MLIREAD           CE MLISETMARK      0BEB NAMEBUFFER      0848 NEXTENTRY
   24 OENTBLK           23 OENTLEN         0879 OPENDIR         08B2 OPENDONE
 09D4 OPENNAME        09D3 OPENPARMS       09D8 OREFNUM           00 OTYPE
 FDDA PRBYTE          09AC PRINTENTRY      09C2 PRNTLOOP        0829 READDIR1
 081F READDIR         098C READNEXT        09D9 READPARMS       08C9 RECURSDIR
 08F7 REOPENED       ?09DD REQAMT         ?09DF RETAMT          09DA RREFNUM
 09E3 SETMPARMS         A0 SPACE           09BB SPCDONE         09B3 SPCLOOP
 0963 SRCPTR          09E4 SREFNUM        ?0800 START           09CF THISBENTRY
 09D0 THISBLOCK       08BA VISITDIR        08B3 VISITFILE
** SUCCESSFUL ASSEMBLY := NO ERRORS
** ASSEMBLER CREATED ON 15-JAN-84 21:28
** TOTAL LINES ASSEMBLED   544
** FREE SPACE PAGE COUNT   81
```

Further Reference

- *ProDOS 8 Technical Reference Manual*
- *AppleShare Programmer's Guide to the Apple IIGS*

ProDOS 8
#18: /RAM Memory Map

Revised by: Matt Deatherage November 1988
Written by: Pete McDonald December 1986

This Technical Note describes the block to actual memory location mapping of /RAM.

Blocks	Address Range	
$70-$7F	$E000-$FFFF	
$68-$6F	$D000-$DFFF	(Bank 2)
$60-$67	$D000-$DFFF	(Bank 1)
$4E-$5C	$A200-$BFFF	
$3D-$4C	$8200-$A1FF	
$2C-$3B	$6200-$81FF	
$1B-$2A	$4200-$61FF	
$0A-$19	$2200-$41FF	

$5D-$5F	$1A00-$1FFF
$4D	$1800-$19FF
$3C	$1600-$17FF
$2B	$1400-$15FF
$1A	$1200-$13FF
$09	$1000-$11FF
$08	$2000-$21FF
$02	$0E00-$0FFF

$03	Bitmap (synthesized)

Notes:

1. Blocks 0, 1, 4, 5, 6, and 7 do not exist.
2. Block $7F contains the Reset, IRQ, and NMI vectors and is normally marked as used.
3. The memory from $0C00 – $0DFF is a general purpose buffer used by the /RAM driver.

Apple II
Technical Notes

Developer Technical Support

ProDOS 8
#19: File Auxiliary Type Assignment

Revised by: Matt Deatherage November 1988
Written by: Matt Deatherage May 1988

This Technical Note describes file auxiliary type assignments.

The information in a ProDOS file auxiliary type field depends upon its primary file type. For example, the auxiliary type field for a text file (TXT, $04) is defined as the record length of the file if it is a random-access file, or zero if it is a sequential file. The auxiliary type field for an AppleWorks™ file contains information about the case of letters in the filename (see Apple II File Type Notes, File Types $19, $1A, and $1B). The auxiliary type field for a binary file (BIN, $06) contains the loading address of the file, if one exists.

Auxiliary types are now used to extend the limit of 256 file types in ProDOS. Specific auxiliary types can be assigned to generic application file types. For example, if you need a file type for your word-processing program, Apple might assign you an auxiliary type for the generic file type of Apple II word processor file, if it is appropriate.

An application can determine if a given file belongs to it by checking the file type and the auxiliary type in the directory entry. Other programming considerations include the following:

1. If your program displays auxiliary type information, it should include all auxiliary types, not just selected ones. Try to display the auxiliary type information stored in the directory entry, just as you would display hex codes for file types for which you do not have a more descriptive message to display.
2. Programs should **not** store information in an undefined auxiliary type field. Storing the record length in a text file is fine, and it is even encouraged, but storing the number of words in a text file in that text file's auxiliary type field might cause problems for those programs which expect to find a record length there. Similarly, storing data in the auxiliary type field will cause problems if your data matches an auxiliary type which is assigned. To avoid these problems, only store defined items in a file's auxiliary type field. If you do not know of a definition for a particular file type's associated auxiliary type, do not store anything in its field.

To request a file type and auxiliary type, please send Apple II Developer Technical Support a description of your proposed file format, along with a justification for not using existing file and auxiliary types. We will publish this information publicly, unless you specifically prohibit it,

since we feel doing so enables the exchange of data for those applications who choose to support other file formats.

Further Reference

- *ProDOS 8 Technical Reference Manual*
- *ProDOS 16 Technical Reference*

ProDOS 8
#20: Mirrored Devices and SmartPort

Revised by: Matt Deatherage November 1988
Written by: Matt Deatherage May 1988

This Technical Note describes how ProDOS 8 reacts when more than two SmartPort devices are connected, how applications using direct device access should behave, and other related issues. This Note supersedes Section 6.3.1 of the *ProDOS 8 Technical Reference Manual*.

Although SmartPort theoretically can handle up to 127 devices connected to a single interface (in practice, electrical considerations curtail this considerably), ProDOS 8 can handle only two devices per slot. This is because ProDOS uses bit 7 of its `unit_number` is used to distinguish drives from each other, and a single bit cannot distinguish more than two devices.

When it boots, ProDOS checks each interface card (or firmware equivalent in the IIc or IIGS) for the ProDOS block-device signature bytes ($Cn01 = $20, $Cn03 = $00, and $Cn05 = $03), so it can install the appropriate device-driver address in the system global page. If the signature bytes match, ProDOS then checks the SmartPort signature byte ($Cn07 = $00), and if that byte matches and the interface is in slot 5 (or located at $C500 in the IIc or IIGS), ProDOS does a SmartPort `STATUS` call to determine how many devices are connected to the interface. If only one or two drives are connected to the interface, ProDOS installs its block-device entry point (the contents of $CnFF added to $Cn00) in the device-driver vector table, which starts at $BF10. In this particular instance, ProDOS would put the vector at $BF1A for slot 5, drive 1, and if two drives were found, at $BF2A for slot 5, drive 2 .

If the interface is in slot 5 and more than two devices are connected, ProDOS copies the same block-device entry point that it uses for slot 5, drives 1 and 2 in the device driver table entry for slot 2, drive 1, and if four drives are connected, for slot 2, drive 2. Further in the boot process, if ProDOS finds the interface of a block device in slot 2 (not possible on a IIc), it replaces the vectors copied from slot 5 with the proper device-driver vectors for slot 2; this is the reason mirroring is disabled if there is a ProDOS device in slot 2. Note that non-ProDOS devices (i.e, serial cards and ports, etc.) do not have vectors installed in the ProDOS device-driver table, so they do not interfere with mirroring.

When ProDOS makes an MLI call with the `unit_number` of a mirrored device, it sets up the call to the device driver then goes through the vector in the device-driver table starting at $BF00. When the block device driver (located on the interface card or in the firmware) gets this MLI call, it checks the unit number which is stored at $43 and verifies if the slot number (bits four, five, and six) is the same as that of the interface. If it is not, the ProDOS block device driver of

the interface realizes it is dealing with a mirrored device, internally adds three to the slot number and two to the drive number, then processes it, returning the desired information or data to ProDOS.

If an application must make direct device-driver calls (something which is **not** encouraged), it should first check `devlst` (starting at $BF32) to verify that the `unit_number` is from an active device. In addition, the application should mask off or ignore the low nibble of entries in `devlst` and know that one less than the number of devices in the list is stored at $BF31 (`devcnt`). The application then should use the `unit_number` to get the proper device-driver vector from the ProDOS global page; the application should **not** construct the vector itself, because this vector would be invalid for a mirrored device.

The following code fragment correctly illustrates this technique. It is written in 6502 assembly language and assumes the `unit_number` is in the accumulator.

```
devcnt      equ   $BF31
devlst      equ   $BF32
devadr      equ   $BF10
devget      sta   unitno                  ; store for later compare instruction
            ldx   devcnt                  ; get count-1 from $BF31
devloop     lda   devlst,x                ; get entry in list
            and   #$F0                    ; mask off low byte
devcomp     cmp   unitno                  ; compare to the unit_number we filled
in
            beq   goodnum                 ;
            dex
            bpl   devloop                 ; loop again if still less than $80
            bmi   badunitno               ; error: bad unit number
goodnum     lda   unitno                  ; get good copy of unit_number
            lsr   a                       ; divide it by 8
            lsr   a                       ; (not sixteen because devadr entries
are
            lsr   a                       ; two bytes wide)
            tax
            lda   devadr,x                ; low byte of device driver address
            sta   addr
            lda   devadr+1,x              ; high byte of device driver address
            sta   addr+1
            rts
addr        dw    0                       ; address will be filled in here by
goodnum
unitno      dfb   0                       ; unit number storage
```

Similarly, applications which construct firmware entry points from user input to “slot and drive” questions will not work with mirrored devices. If an application wishes to issue firmware-specific calls to a device, it should look at the high byte of the device-driver table entry for that device to obtain the proper place to check firmware ID bytes. In the sample code above, the high byte would be returned in addr+1. For devices mirrored to slot 2 from slot 5, this technique will return $C5, and ID bytes would then be checked (since they should **always** be checked before making device-specific calls) in the $C500 space. Applications ignoring this technique will incorrectly check the $C200 space.

Further Reference

- *ProDOS 8 Technical Reference Manual*

- ProDOS 8 Technical Note #21, Identifying ProDOS Devices

ProDOS 8
#21: Identifying ProDOS Devices

Revised by: Dave Lyons & Matt Deatherage March 1990
Written by: Matt Deatherage & Dan Strnad November 1988

This Technical Note describes how to identify ProDOS devices and their characteristics given the ProDOS unit number. This scheme should only be used under ProDOS 8.
Changes since January 1990: Modified AppleTalk call code for compatibility with ProDOS 8 versions earlier than 1.5 and network-booted version 1.4.

There are various reasons why an application would want to identify ProDOS devices. Although ProDOS itself takes great pains to treat all devices equally, it has internal drivers for two types of devices: Disk II drives and the /RAM drive provided on 128K or greater machines. Because all devices really are not equal (i.e., some cannot format while others are read-only, etc.), a developer may need to know how to identify a ProDOS device.

Although the question of how much identification is subjective for each developer, ProDOS 8 offers a fair level of identification; the only devices which cannot be conclusively identified are those devices with RAM-based drivers, and they could be anything. The vast majority of ProDOS devices can be identified, however, so you could prompt the user to insert a disk in UniDisk 3.5 #2, instead of Slot 2, Drive 2, which could be confusing if the user has a IIc or IIGS.

Note that for the majority of applications, this level of identification is unnecessary. Most applications simply prompt the user to insert a disk by its name, and the user can place it in any drive which is capable of working with the media of the disk. You should avoid requiring a certain disk to be in a specific drive since doing so defeats much of the device-independence which gives ProDOS 8 its strength.

When you do need to identify a device (i.e., if you need to format media in a Disk II or /RAM device), however, the process is fairly straightforward. This process consists of a series of tests, any one of which could end with a conclusive device identification. It is not possible to look at a single ID byte to determine a particular device type. You may determine rather quickly that a device is a SmartPort device, or you may go all the way through the procedure to identify a third-party network device. For those developers who absolutely must identify devices, DTS presents the following discussion.

Isn't There Some Kind of "ID Nibble?"

ProDOS 8 does not support an "ID nibble." Section 5.2.4 of the *ProDOS 8 Technical Reference Manual* states that the low nibble of each unit number in the device list "is a device identification: 0 = Disk II, 4 = Profile, $F = /RAM."

When ProDOS 8 finds a "smart" ProDOS block device while doing its search of the slots and ports, it copies the high nibble of $CnFE (where n is the slot number) into the low nibble of the unit number in the global page. The low nibble then has the following definition:

Bit 3: Medium is removable
Bit 2: Device is interruptible
Bit 1-0: Number of volumes on the device (minus one)

As you can see, it is quite easy for the second definition to produce one of the original values (e.g., 0, 4, or $F) in the same nibble for completely different reasons. You should ignore the low nibble in the unit number in the global page when identifying devices since the first definition is insufficient to uniquely identify devices and the second definition contains no information to specifically identify devices. Once you do identify a ProDOS block device, however, you may look at $CnFE to obtain the information in the second definition above, as well as information on reading, writing, formatting, and status availability.

When identifying ProDOS devices, start with a list of unit numbers for all currently installed disk devices. As you progress through the identification process, you identify some devices immediately, while others must wait until the end of the process for identification.

Starting with the Unit Number

ProDOS unit numbers (`unit_number`) are bytes where the bits are arranged in the pattern DSSS0000, where D = 0 for drive one and D = 1 for drive two, SSS is a three-bit integer with values from one through seven indicating the device slot number (zero is not a valid slot number), and the low nibble is ignored.

To obtain a list of the unit numbers for all currently installed ProDOS disk devices, you can perform a ProDOS MLI `ON_LINE` call with a unit number of $00. This call returns a unit number and a volume name for every device in the device list. ProDOS stores the length of the volume name in the low nibble of the unit number which `ON_LINE` returns; if an error occurs, the low nibble contains $0 and the byte immediately following the unit number contains an error code. For more information on the `ON_LINE` call, see section 4.4.6 of the *ProDOS 8 Technical Reference Manual*. A more detailed discussion of the error codes follows later in this Note.

To identify the devices in the device list, you need to know in which physical slot the hardware resides, so you can look at the slot I/O ROM space and check the device's identification bytes. Note that the slot-number portion of the unit number does not always represent the physical slot of the device, rather, it sometimes represents the logical slot where you can find the address of the device's driver entry point in the ProDOS global page. For example, if a SmartPort device interface in slot 5 has more than two connected devices, the third and fourth devices are mapped to slot 2; this mapping gives these two devices unit numbers of $20 and $A0 respectively, but the device's driver entry point is still in the $C5xx address space.

ProDOS 8 Technical Note #20, Mirrored Devices and SmartPort, discusses this kind of mapping in detail. It also presents a code example which gives you the correct device-driver entry point

(from the global page) given the unit number as input. Here is the code example from that Note for your benefit. It assumes the `unit_number` is in the accumulator.

```
devcnt      equ   $BF31
devlst      equ   $BF32
devadr      equ   $BF10
devget      sta   unitno            ; store for later compare instruction
            ldx   devcnt            ; get count-1 from $BF31
devloop     lda   devlst,x          ; get entry in list
            and   #$F0              ; mask off low nibble
devcomp     cmp   unitno            ; compare to the unit_number we filled in
            beq   goodnum           ;
            dex
            bpl   devloop           ; loop again if still less than $80
            bmi   badunitno         ; error: bad unit number
goodnum     lda   unitno            ; get good copy of unit_number
            lsr   a                 ; divide it by 8
            lsr   a                 ; (not sixteen because devadr entries are
            lsr   a                 ; two bytes wide)
            tax
            lda   devadr,x          ; low byte of device driver address
            sta   addr
            lda   devadr+1,x        ; high byte of device driver address
            sta   addr+1
            rts
addr        dw    0                 ; address will be filled in here by goodnum
unitno      dfb   0                 ; unit number storage
```

Warning: Attempting to construct the device-driver entry point from the unit number is very dangerous. **Always** use the technique presented above.

Network Volumes

AppleTalk volumes present a special problem to some developers since they appear as "phantom devices," or devices which do not always have a device driver installed in the ProDOS global page. Fortunately, the ProDOS Filing Interface (PFI) to AppleTalk provides a way to identify network volumes through an MLI call. The ProDOS Filing Interface call `FIListSessions` is used to retrieve a list of the current sessions being maintained through PFI and any volumes mounted for those sessions.

In the following example, note the check for ProDOS 8 version 1.5 or higher, and the simulation of a `bad command` error under older versions (the $42 call under ProDOS 8 version 1.4 always crashes if ProDOS was launched from a local disk):

```
Network     LDA   #$04              ;require at least ProDOS 8 1.4
            CMP   $BFFF             ;KVERSION (ProDOS 8 version)
            BEQ   MoreNetwork       ;have to check further
            LDA   #$01              ;simulate bad command error
            BCS   ERROR             ;if 3 or less, no possibility of network
            BCC   NetCall           ;otherwise, try the network call

MoreNetwork LDA   $BF02             ;high byte of the MLI entry point
            AND   #$F0              ;strip off the low nibble
            CMP   #$C0              ;is the entry into the $Cn00 space?
            BEQ    NetCall          ;yes, so try AppleTalk
            LDA   #$01
            SEC
            BCS   ERROR             ;simulate bad command error

NetCall     JSR   $BF00             ;ProDOS MLI
```

```
             DFB    $42                ;AppleTalk command number
             DW     ParamAddr          ;Address of Parameter Table
             BCS    ERROR              ;error occurred

ParamAddr    DFB    $00                ;Async Flag (0 means synchronous only)
                                       ;note there is no parameter count
             DFB    $2F                ;command for FIListSessions
             DW     $0000              ;AppleTalk Result Code returned here
             DW     BufLength          ;length of the buffer supplied
             DW     BufPointer         ;low word of pointer to buffer
             DW     $0000              ;high word of pointer to buffer
                                       ;(THIS WILL NOT BE ZERO IF THE BUFFER IS
                                       ;NOT IN BANK ZERO!)
             DFB    $00                ;Number of entries returned here
```

If the `FIListSessions` call fails with a bad command error ($01), then AppleShare is not installed; therefore, there are no networks volumes mounted. If there is a network error, the accumulator contains $88 (Network Error), and the result code in the parameter block contains the specific error code. The list of current sessions is placed into the buffer (at the address `BufPointer` in the example above), but if the buffer is not large enough to hold the list, it retains the maximum number of current sessions possible and returns an error with a result code of $0A0B (`Buffer Too Small`). The buffer format is as follows:

```
SesnRef      DFB    $00                ;Sessions Reference number (result)
UnitNum      DFB    $00                ;Unit Number (result)
VolName      DS     28                 ;28 byte space for Volume Name
                                       ;(starts with a length byte)
VolumeID     DW     $0000              ;Volume ID (result)
```

This list is repeated for every volume mounted for each session (the number is placed into the last byte of the parameter list you passed to the ProDOS MLI). For example, if there are two volumes mounted for session one, then session one is listed two times. The `UnitNum` field contains the slot and drive number in unit-number format, and note that bit zero of this byte is set if the volume is a user volume (i.e., it contains a special “users” folder). This distinction is unimportant for identifying a ProDOS device as a network pseudo-device, but it is necessary for applications which need to know the location of the user volume. Note that if you mount two servers or more with each having its own user volume, the user volume found first in the list (scanned top to bottom) returned by `FIListSessions` specifies the user volume that an application should use. See the *AppleShare Programmer’s Guide for the Apple IIGS* for more information on programming for network volumes.

If you keep a list of all unit numbers returned by the `ON_LINE` call and mark each one “identified” as you identify it, keep in mind that the unit numbers returned by `FIListSessions` and `ON_LINE` have different low nibbles which should be masked off before you make any comparisons.

Note: You should mark the network volumes as identified and **not** try to identify them further with the following methods.

What Slot is it Really In?

Once you have the address of the device driver's entry point and know that the device is not a network pseudo-device, you can determine in what physical slot the device resides. If the high byte of the device driver's entry point is of the form $Cn, then n is the physical slot number of the device. A SmartPort device mirrored to slot 2 has a device driver address of $C5xx, giving 5 as the physical slot number.

If the high byte of the device driver entry point is **not** of the form $Cn, then there are three other possibilities:

- The device is a Disk II with driver code inside ProDOS.
- The device is either /RAM with driver code inside ProDOS or a third-party auxiliary-slot RAM disk device with driver code installed somewhere in memory.
- The device is not a RAM disk but has a RAM-based device driver, like a third-party network device.

Auxiliary-slot RAM disks are identified by convention. Any device in slot 3, drive 2 (unit number $B0) is assumed to be an auxiliary-slot RAM disk since ProDOS 8 does not recognize any card which is not an 80-column card in slot 3 (see ProDOS 8 Technical Note #15, How ProDOS 8 Treats Slot 3). There is a chance that some other kind of device could be installed with unit number $B0, but it is not likely.

To identify various kinds of auxiliary-slot RAM disks, you must obtain the unit number from the ProDOS global page. The list of unit numbers starts at $BF32 (`DEVLST`) and is preceded by the number of unit numbers minus one (`DEVCNT`, at $BF31). You should search through this list until you find a unit number in the form $Bx; if the unit number is $B3, $B7, $BB, or $BF, you can assume the device to be an auxiliary-slot RAM disk which uses the auxiliary 64K bank of memory present in a 128K Apple IIe or IIc, or a IIGS. If the unit number is one of the four listed above, you must remove this device to safely access memory in the auxiliary 64K bank, but if the unit number is **not** one of the four listed above, you can assume the device to be an auxiliary-slot RAM disk which does **not** use the normal bank of auxiliary memory. (Some third-party auxiliary-slot cards contain more than one 64K auxiliary bank; the normal use of this memory is as a RAM disk. If the RAM-based driver for this kind of card does not use the normal auxiliary 64K bank for storage, it should have a unit number other than one of the four listed above.) If the unit number is not one of the four listed above, you may safely access the auxiliary bank of memory without first removing this device.

Section 5.2.2.3 of the *ProDOS 8 Technical Reference Manual* contains a routine which disconnects the appropriate RAM disk devices in slot 3, drive 2, without removing those drivers which do not use that bank, to allow use of the auxiliary 64K bank.

Note: Previous information from Apple indicated that /RAM could be distinguished from third-party RAM disks by a driver address of $FF00. Although the address has not changed, some third-party drivers may have addresses of $FF00 as well, although this is **not** supported. /RAM always has a driver address of $FF00 and unit number $BF, although any third-party RAM disk could install itself with similar attributes.

For Disk II devices, the three-bit slot number portion of the `unit_number` is **always** the physical slot number. Disk II devices can never be mirrored to another slot (the Disk II driver does not support it); therefore, it is in the physical slot represented in the unit number which ProDOS assigns when it boots.

If the high byte of the device driver's entry point is not of the form $Cn, then you should assume that the slot number is the value SSS in the unit number (this is equivalent to assuming the device is a Disk II) for the next step, which is checking the I/O space for identification bytes.

What to Do With the Slot Number

Once you have the slot number, you can look at the slot I/O ROM space to determine the kind of device it is. As described in the *ProDOS 8 Technical Reference Manual*, ProDOS looks for the following ID bytes in ROM to determine if a ProDOS device is in a slot:

$Cn01 = $20
$Cn03 = $00
$Cn05 = $03

If you use the slot number, n, you obtained above, and the three values listed above are **not** present, then the device has a RAM-based driver and cannot further be identified.

If the three values previously discussed are present, then examination of $CnFF gives more information. If $CnFF = $00, the device is a Disk II. If $CnFF is any value other than $00 or $FF ($FF signifies a 13-sector Disk II, which ProDOS does not support), the device is a ProDOS block device.

For ProDOS block devices, the byte at $CnFE contains several flags which further identify the device; these flags are discussed in section 6.3.1 of the *ProDOS 8 Technical Reference Manual.*

SmartPort Devices

Many of Apple's ProDOS block devices follow the SmartPort firmware interface. Through SmartPort, you can further identify devices. Existing SmartPort devices include SCSI hard disks, 3.5" disk drives and CD-ROM drives, with many more possible device types.

If $Cn07 = $00, then the device is a SmartPort device, and you can then make a SmartPort call to get more information about the device, including a device type and subtype. The SmartPort entry point is three bytes beyond the ProDOS block device entry point, which you already determined. The method for making SmartPort calls is outlined in the *Apple IIc Technical Reference Manual, Second Edition* and the *Apple IIGS Firmware Reference*.

The most useful SmartPort call to make for device identification is the `STATUS` call with `statcode` = 3 for Return Device Information Block (DIB). This call returns the ASCII name of the device, a device type and subtype, as well as the size of the device. Some SmartPort device types and subtypes are listed in the referenced manuals, with a more complete list located in the *Apple IIGS Firmware Reference*. A list containing SmartPort device types only is provided in SmartPort Technical Note #4, SmartPort Device Types.

RAM-Based Drivers

One fork of the identification tree comes to an end at this point. If the high byte of the device driver entry point was not $Cn and the device was not /RAM, you assumed it was a Disk II and used the slot number portion of the unit number to examine the slot ROM space. If the ROM space for that slot number does not match the three ProDOS block device ID bytes, it cannot be a Disk II. Having ruled out other possibilities, it must be a device installed after ProDOS finished building its device table. Perhaps it is a third-party RAM disk driver or maybe a driver for an older card which does not match the ProDOS block device ID bytes.

Whatever the function of the driver, you can identify it no further. It quite literally could be any kind of device at all, and with neither slot ROM space to identify nor a standard location to compare the device driver entry point against, the best you can do is consider it a "generic device" and go on.

But Is It Connected and Can I Read From It?

Just because a ProDOS device is in the table does not mean it is ready to be used. There is always the possibility that the drive has no media in it. Back in the beginning, you made an `ON_LINE` call with a unit number of $00. If the volume name of a disk in that device could not be read, or another error occurred, ProDOS 8 would return the error code in the `ON_LINE` buffer immediately following the unit number. Those errors possible include:

$27	I/O error
$28	No Device Connected
$2B	Write Protected
$2F	Device off-line
$45	Volume directory not found
$52	Not a ProDOS disk
$55	Volume Control Block full
$56	Bad buffer address
$57	Duplicate volume on-line

Note that error $2F is not listed in the *ProDOS 8 Technical Reference Manual.*

By convention, you interpret I/O error to mean the disk in the drive is either damaged or blank (not formatted). You interpret Device off-line to mean that there is no disk in the drive. You interpret No Device Connected to mean the drive really does not exist (for example, asking for status on a second Disk II when only one is connected).

If no error occurred for a unit number in the `ON_LINE` call (the low nibble of the unit number is not zero), the volume name of the disk in the drive follows the unit number.

Things To Avoid

The ProDOS device-level `STATUS` call generally returns the number of blocks on a device. Applications should **not** try to identify 3.5" drives by doing a ProDOS or SmartPort `STATUS` call and comparing the number of blocks to 800 or 1,600. The correct way to identify a 3.5" drive is by the `Type` field in a SmartPort `STATUS` call.

Don't assume the characteristics of a device just because it is in a certain slot. For example, be prepared to deal with 5.25" disk drives in slots other than 6. Don't assume that slot 6 is associated with block devices at all—there could be a printer card installed.

Avoid reinstalling /RAM when your application finds it removed. If you remove /RAM, you should reinstall it when you're done with the extra memory; however, if your application finds /RAM already gone, you do **not** have the right to just reinstall it. A driver of some kind may be installed in auxiliary memory, and arbitrary reinstallation of /RAM could bring the system down.

Further Reference

- *ProDOS 8 Technical Reference Manual*
- *AppleShare Programmer's Guide for the Apple IIGS* (APDA)
- ProDOS 8 Technical Note #15, How ProDOS 8 Treats Slot 3
- ProDOS 8 Technical Note #20, Mirrored Devices and SmartPort
- ProDOS 8 Technical Note #23, ProDOS 8 Changes and Minutia
- ProDOS 8 Technical Note #26, Polite Use of Auxiliary Memory

Apple II
Technical Notes

Developer Technical Support

ProDOS 8
#22: Don't Put Parameter Blocks on Zero Page

Written by: Dave Lyons July 1989

Putting ProDOS 8 parameter blocks on zero page ($00-$FF) is not recommended.

It is not a good idea to put the parameter blocks for ProDOS 8 MLI calls on zero page. This is not forbidden by the *ProDOS 8 Technical Reference Manual*, but then again, it also doesn't tell you not to put parameter blocks in ROM, in the $C0xx soft switch area, or just below the active part of the stack.

If you **do** put MLI parameter blocks on zero page, your application may break in the future.

If your parameter block comes between $80 and $FF, it won't work with AppleShare installed.

Further Reference

- *ProDOS 8 Technical Reference Manual*

ProDOS 8
#23: ProDOS 8 Changes and Minutia

Revised by: Matt Deatherage May 1992
Written by: Matt Deatherage July 1989

This Technical Note documents the change history of ProDOS 8 through V2.0.1, and it supersedes the information on this topic in the *ProDOS 8 Technical Reference Manual* and the *ProDOS 8 Update*.
Changes since September 1990: Updated to include ProDOS 8 version 2.0.1 and its known bugs. Replaced APDA references with Resource Central .

Changes? You're kidding.

No. One of the side effects of evolving technology is that eventually little things (like the disk operating system) have to change to support the new technologies. Every time Apple changes ProDOS 8, the manuals can't be reprinted. For one thing, it takes a long time to turn out a manual, by which time there's often a new version done which the new manual doesn't cover. For another thing, programmers and developers don't tend to purchase revised manuals (our informal research shows that more people have up-to-date Apple /// RPS documentation than have up-to-date Apple IIc documentation—and this was done before the Apple IIc Plus was released…).

So this Note explains what has changed between ProDOS 8 V1.0 and the current release, V2.0.1, which began shipping with Apple IIGS System Software 6.0. Table 1 shows what versions of ProDOS 8 existing documentation covers.

Document	Version Number
ProDOS 8 Technical Reference Manual	1.1.1
ProDOS 8 Update	1.4
AppleShare Programmer's Guide to the Apple IIGS	1.5

Table 1–ProDOS 8 Documentation

ProDOS 1.0

This was the first release of ProDOS, which was so unique it didn't even have to be called ProDOS 8 to distinguish it from ProDOS 16. If you have documentation that predates ProDOS 1.0, you should seek professional help from Resource Central at the address listed in Technical Note #0.

ProDOS 1.0.1

- Fixed a bug in the `STATUS` call which affected testing for the write-protected condition.

ProDOS 1.0.2

- Changed instructions used in interrupt entry routines on the global page so the accumulator would not be destroyed.
- Fixed a bug in the Disk II core routines so the motor would shut off after recalibration on an error.

ProDOS 1.1

- Changed the internal MLI layout for future expansibility and maintenance.
- Modified machine ID routines to identify IIc and enhanced IIe ROMs.
- Removed code that allowed ProDOS to boot on 48K machines.
- Removed the check for the ProDOS version number from the `OPEN` routine.
- Incremented `KVERSION` (the ProDOS Kernel version) on the global page.
- Modified the loader routines to reflect the presence of any 80-column card following the established protocol (see ProDOS 8 Technical Note #15, How ProDOS 8 Treats Slot 3). Also, at this time, added code to allow slot 3 to be enabled on a IIe if an 80-column card following the protocol was found.
- Added code to turn off all disk motor phases prior to seeking a track in the Disk II driver.
- Fixed a bug to prevent accesses to /RAM after it had been removed from the device list.
- Reduced the size of the /RAM device by one block to protect interrupt vectors in the auxiliary language card. The correct vectors are installed at boot time.

ProDOS 1.1.1

- Fixed a Disk II driver bug for mapping into drive 1.
- Modified machine ID routines to give precedence to identifiable 80-column cards in slot 3.

ProDOS 8 1.2

- Changed the name from ProDOS to ProDOS 8 to avoid confusion with ProDOS 16, which, again, this Note does not discuss.
- Introduced the clock driver for the Apple IIGS. The machine identification code was changed to indicate the presence of the clock on the IIGS.
- Added preliminary network support by adding the network call and preliminary network driver space.
- Fixed a bug in returning errors from calls to the RAM disk. Changed the RAM disk driver to return values of zero on reads and ignore writes to blocks zero, one, four, five, six, and seven, which are not accessible as storage in the driver's design.
- Added a new system error ($C) for errors when deallocating blocks from a tree file.
- Fixed a bug in zeroing a Volume Control Block (VCB) when trying to reallocate a previously used VCB.
- Modified the ProDOS 8 loader code to automatically install up to four drives in slot 5 if a SmartPort device is found. Removed the code to always leave interrupts disabled, which leaves the state of the interrupt flag at boot time unchanged while ProDOS 8 loads.

- Changed the MLI entry to disable interrupts until after the `MLIACTV` flag is set and other ProDOS parameters are initialized.
- Modified the `QUIT` code to allow the Delete key to function the same as the left arrow key. Also fixed a bug so screen holes would not be trashed in 80-column mode. Crunched code to allow soft switch accesses to force 40-column text mode. Fixed a bug so the dispatcher would not trash the screen when executed with a `NIL` prefix.
- Modified the `ONLINE` call so that it could be made to a device that had just been removed from the device list by the standard protocol. Previous to this change, a VCB for the removed device was left, reducing the number of on-line volumes by one for each such device. From this point on, removing a device should be followed by an `ONLINE` call to the device just removed. The call returns error $28 (`No Device Connected`), but deallocates the VCB.
- Added a spurious interrupt handler to allow up to 255 unclaimed interrupts before system death.
- Removed the code which invoked low-resolution graphics on system death—it had not worked well and the space was needed. The system had previously had the ability to display "`INSERT SYSTEM DISK AND RESTART`" without also displaying "`-ERR xx`", which was removed at this point for space reasons since the system wasn't using it (and hopefully you weren't, either, since it wasn't documented).
- Changed `MLIACTV` to use an `ASL` instead of an `LSR` to turn "off" the flag.
- Changed the `OPEN` call to correctly return error $4B (`Unsupported Storage Type`) instead of error $4A (incompatible file format for this version) when attempting to open a file with an unrecognized storage type.
- Fixed an obscure bug involving `READ` in Newline mode. If the requested number of bytes was greater than $FF, **and** the number of bytes in the file **after** the newline character was read was a multiple of $100, then the number of bytes reported transferred by ProDOS was equal to the correct number of transferred bytes plus $100.
- Starting with V1.2 on an Apple IIGS, stopped switching slot 3 ROM space and left the determination of whether the slot or the port was enabled to the Control Panel; however, there was a bug in this implementation which was fixed in V1.7 and described in ProDOS 8 Technical Note #15, How ProDOS 8 Treats Slot 3.
- Updated the slot-based clock driver's year table through 1991.
- Added a feature which allows ProDOS 8 to search for a file named ATINIT in the boot volume's root directory, to load and execute it, then to proceed normally with the boot process by loading the first .SYSTEM file. No error occurs if the ATINIT file is not found, but any other error condition (including the file existing and not having file type $E2) causes a fatal error.
- Changed loader code so ProDOS 8 could be loaded by ProDOS 16 without automatically executing the ATINIT and the first .SYSTEM file.
- Changed the device search process in the ProDOS 8 loader so SmartPort devices are only installed if they actually exist, and Disk IIs are placed with lowest priority in the device list so they are scanned last.
- Forced Super Hi-Res off on an Apple IIGS when a fatal error occurs. (Actually, this did not work, but it was fixed in V1.7.)
- Inserted a patch to fix a bug in the first IIGS ROM that caused internal $Cn00 ROM space to be left mapped in if SmartPort failed to boot.

ProDOS 8 1.3

Warning: This is not a stable version of ProDOS due to an illegal 65C02 instruction which was added. This version can damage disks if used with a 6502 processor.

- Changed the code that resets phase lines for Disk IIs so phase clearing is done with a load instead of a store, since stores to even numbered locations cause bus contention, which is major uncool. Changed the routine to force access to all eight even locations, which not only clears the phases, but also forces read mode, first drive, and motor off. DOS used to do this; ProDOS had not been doing it. If `L7` had been left on when the Disk II driver was called and it checked write-protect with `L6` high, write mode was enabled. Forcing read mode leaves less to chance.
- Changed deallocation of index blocks so index blocks are not zeroed, allowing the use of file recovery utilities. Instead, index blocks are "flipped" (the first 256 bytes are exchanged with the last 256 bytes).
- Since the UniDisk 3.5 interface card for the][+ and IIe does not set up its device chain unless a ProDOS call is made to it, ProDOS `STATUS` calls are now made to the device before SmartPort `STATUS` calls.

ProDOS 8 1.4

- Removed an illegal 65C02 instruction which was added in V1.3.
- Modified the Disk II driver so a routine that should only clear the phase lines only clears the phase lines. Also clear `Q7` to prevent inadvertent writes.

Warning: The AppleTalk command, which was added in version 1.5, is present as a skeleton in this version. Unfortunately, it's not a useful skeleton. It moves a section of memory from a ProDOS location to another location and transfers control, totally oblivious of the fact that there is no code at this address.

Even more unfortunate, the server software that ships with the Apple IIe Workstation Card is such that when the IIe is booted over the network with that server software, it is version 1.4 (`KVERSION = 4`).

So if you boot version 1.4 from a local disk, making a $42 call is fatal. See ProDOS 8 Technical Note #21, Identifying ProDOS Devices, for a reliable way to identify AppleTalk volumes under ProDOS 8 version 1.4.

ProDOS 8 1.5

- ProDOS 8 1.5 is the first version to include network support through the ProDOS Filing Interface (PFI) as part of ProDOS 16 or on the Apple IIe Workstation Card without booting over the server (see the warning under version 1.4). Made many changes to internal routines for PFI location and compatibility at this point. Crunched and moved code for PFI booting and accessibility.
- Changed some strings to all uppercase internally for string comparisons.
- Removed the generic $42 AppleTalk call (the cause of the previous warning) which was introduced in V1.2, as PFI gets called through the global page.
- Changed the `ASL` to clear the `MLIACTV` flag back to an `LSR`. This doesn't make nested levels of busy states possible, but always clears the flag before calling interrupt handling routines that check `MLIACTV` as described in the *ProDOS 8 Technical Reference Manual*.

- If an Escape key is detected in the keyboard buffer on an Apple IIc, it is removed. This is friendly to the Apple IIc Plus, the ROM of which does not remove the Escape key it uses to detect that the system should be booted at normal speed.

ProDOS 8 1.6

- Set up a parallel pointer to correct a PFI misinterpretation of an internal MLI pointer.

ProDOS 8 1.7

- Made a change to ensure that ProDOS 8 counts the volume's bitmap before incrementing the number of free blocks. This fixed a bug where an uninitialized location was being incremented and decremented, incorrectly reporting a `Disk Full` error where none should have occurred.
- Changed the handling of slot 3 ROM space to that described in ProDOS 8 Technical Note #15, How ProDOS 8 Treats Slot 3.
- Changed code to permit the invisible bit of the access byte (bit 2) to be set by applications.

ProDOS 8 1.8

- Fixed a bug introduced in V1.3. If an error occurs while calling `DESTROY` on a file, the file is not deleted but the index blocks are not swapped back to normal position. If a subsequent `DESTROY` of the same file succeeds, the volume's integrity is destroyed. Now ProDOS 8 marks the file as deleted, even if an error occurs, so any other errors do not cause a subsequent MLI call to trash the volume. Note that "undelete" utilities attempting to undelete such a file (one in which an error occurred during the `DESTROY`) may **trash** the volume.
- Fixed the `ONLINE` call to ignore the unused low nibble of the `unit_num` parameter when deciding how many bytes to zero in the application's buffer. This change fixes a bug which zeroed only the first 16 bytes of the caller's buffer before filling them if an `ONLINE` call was made with a `unit_num` of $0X, where X is non-zero.
- When loading on an Apple IIGS, ProDOS 8 now sets the video mode so the 80-column firmware is not active when the ProDOS 8 application gets control.
- Changed internal version checking between GS/OS and ProDOS 8. Note that GS/OS and ProDOS 8 are still tied to each other—versions that didn't come on the same disk can't be used together. The methods for checking versions were just altered.
- Made the backward compatibility check when opening subdirectories inactive. The test would always fail when opening a subdirectory with lowercase characters in the name (as assigned by the ProDOS FST under GS/OS), so the check was removed. Note that using earlier versions of ProDOS 8 with such disks causes errors when trying to access files with such directories in their pathnames.
- Expanded the ProDOS 8 loader code to provide for more room for future compatibility.
- On a IIGS, installs a patch into the GS/OS stack-based call vector so that anyone calling GS/OS routines (like `QDStartUp` in ROM 03, for example) gets an appropriate error instead of performing a `JSL` into the stratosphere.

ProDOS 8 1.9

- New selector and dispatcher code was added for machines with 80 columns. The old code is still present for machines without 80-column capability.

- Fixed two bugs involved in booting into a ".SYSTEM" program larger than 38K. First, ProDOS 8 should be able to boot into a program as large as 39.75K, but was returning an error if the ".SYSTEM" program was larger than 38K. Second, when attempting to print the message "`*** SYSTEM PROGRAM TOO LARGE ***`", only one asterisk was printed. Both these bugs are fixed.

- No longer requires a ".SYSTEM" file when booting. If ProDOS 8 does not find a ".SYSTEM" file and the enhanced selector and dispatcher code is installed, ProDOS 8 executes a `QUIT` call.

- `KVERSION` is still $08.

ProDOS 8 v2.0.1

- ProDOS 8 now supports more than two SmartPort devices per slot by remapping the third device and beyond to different slots. There's still a limit of 14 devices altogether, though.

- ProDOS 8 version 2.0.1 and later **require** a 65C02 microprocessor or equivalent; you get `RELOCATION/CONFIGURATION ERROR` if you don't have one. ProDOS 8 tests for a 65C02 by setting binary-coded decimal (BCD) mode and adding $01 to $99, which is the largest negative BCD value representable in an 8-bit register. 65C02 microprocessors correctly clear the `N` flag when the addition wraps to zero; 6502 microprocessors do not.

  Since all of Apple's 65C02 or greater computers also have lower-case capability, the ProDOS 8 splash screen now uses lower-case letters. After only nine years, too.

- The file's been rearranged again, so if you have a program that patches the P8 file, it probably breaks now. Please learn your lesson and write a `.SYSTEM` program that patches ProDOS 8 in memory and not on disk.

- The prefix is now set correctly when launching Applesoft programs.

- Old never-used code to handle call $42 is now gone.

- Removed some RAM-disk code that was not used.

- ProDOS 8 now sets the prefix to empty when you try to set the prefix to "`/`".

- The Apple IIGS clock driver inside ProDOS 8 now limits the year to the range 00 through 99.

- Sparse seedling files are now truncated properly.

- When filling up a volume with a `WRITE` call, ProDOS 8 used to return the disk full error but leave the file's mark set past the file's EOF. This is now fixed.

- If you try to mount a new volume but all eight VCB slots are filled, ProDOS 8 now tries to kick out the first volume in the table with no open files. If all volumes have open files, you'll still get error $55.

- The new quit code (introduced with 1.9) now beeps and lets the user try another subdirectory if the one they chose can't be opened. Previously it went forward to the next volume.

- The new quit code also now closes a directory if it gets a ProDOS error in the directory read loop.

- When synthesizing a directory entry for a volume, ProDOS 8 always used to assume the directory was four blocks long (for 51 files). The `/RAM` disk's directory is shorter than this (one block), and ProDOS 8 no longer returns funky errors when trying to read past the end of this shortened directory. The EOF and blocks used are now returned as $200 and 1, respectively.

- The system death messages are now displayed in the center of the 40-column screen, bordered by inverse spaces. This is an improvement over the line of garbage showing at the bottom of the screen since approximately version 1.5.

- The new quit code was rearranged to clear the screen prior to loading the selected application. This insures that `MSLOT` ($07F8) points to the "boot" slot prior to starting the application. In this way, you can launch the `ProDOS` file from GS/OS to start up GS/OS. (Note that `MSLOT` **must** be set properly for this to work.)

- If the device search code at start time finds a SCSI SmartPort, a SmartPort status call is issued to device #2. This lets the Apple II High-Speed SCSI card build its device tables and return the true number of devices connected. Without this, it always returns "4" for slot 5 or "2" for any other slot.

- `KVERSION` is now $21.

Known ProDOS 8 v2.0.1 bugs

- ProDOS 8 still doesn't behave perfectly when 14 or more devices are present. Specifically, the `/RAM` driver tends to install itself without checking to see whether or not there's room in the device table.

Caution: ProDOS 8's remapping of SmartPort devices may interfere with intelligent SmartPort peripherals that were already doing their own remapping. ProDOS 8 remaps additional SmartPort devices, even if the SmartPort firmware already did this on its own, and this can cause problems. We never said this would work, but we never said it wouldn't—ProDOS 8 has no way to determine what remapping has already been done. If you make such a card and your customers have problems, tell them to disable your SmartPort remapping and let ProDOS 8 do it all.

Further Reference

- *ProDOS 8 Technical Reference Manual*
- *ProDOS 8 Update*
- *AppleShare Programmer's Guide to the Apple II*
- ProDOS 8 Technical Note #21, Identifying ProDOS Devices

ProDOS 8
#24: BASIC.SYSTEM Revisions

Revised by: Matt Deatherage May 1992
Written by: Matt Deatherage July 1989

This Technical Note documents the change history of BASIC.SYSTEM through V1.5, which ships with Apple IIGS System Software 6.0. V1.0, the initial release, is not documented in this Note, and V1.1 is described in *BASIC Programming with ProDOS*.
Changes since September 1990: Revised to include BASIC.SYSTEM 1.5.

V1.1

- Fixed a bug in variable packing (used by `CHAIN`, `STORE`, and `RESTORE`).
- Changed the interpreter to use the ProDOS startup convention of a `JMP` instruction followed by two $EE bytes and a startup pathname buffer.
- Removed a bad buffer address in the `FIELD` parameter of the `READ` routine.
- Fixed a bug in `APPEND` so calls to `OPEN` and `READ` from a random-access file would not cause the next call to `APPEND` to any file to use the record length of the random-access file.
- Added the `BYE` command to allow ProDOS `QUIT` calls from BASIC.
- Removed the limited support for run-time capabilities which had been present.

V1.2

- Changed the `CATALOG` command to ignore the number of entries in a directory when listing it so AppleShare volumes could be cataloged properly (this number can change on the fly on an AppleShare volume).
- Fixed another bug in `CATALOG` so pressing an unexpected key when a catalog listing was paused with a Control-S would no longer abort the catalog.

V1.3

- Changed `BSAVE` so it now truncates the length of the saved file when the `B` parameter is not used. To replace the first part of a file without truncation, use the `B` parameter with a value of zero. This behavior with the `B` parameter is how V1.1 and V1.2 worked without the `B` parameter.
- Fixed a bug in `CHAIN` and `STORE` where they expected one branch to go two ways at the same time.
- Added the `MTR` command for easier access to the Monitor from BASIC.
- Made internal changes to the assembly process for easier project management. These changes do not affect the code image.

V1.4

- Fixed a bug which caused a `BLOAD` into an address marked as used in the global page to start performing a `BSAVE` on the file instead of returning the `NO BUFFERS AVAILABLE` message. For this reason, BASIC.SYSTEM version 1.3 should **not** be used.

V1.4.1

- Fixed a bug in the mark handling routines. When using the "B" parameter to indicate a byte to use as a file mark, the third and most significant byte would never be reset before the next use of B. For example, if you used a B value of $010000 and then used a B value of $2345, BASIC.SYSTEM 1.4 would use $012345 for the second B parameter value.

V1.5

- Fixed centuries-old bug where `NOTRACE` after a `THEN` (as in `IF/THEN`) disconnected BASIC.SYSTEM. Now it doesn't.
- `BSAVE` now modifies the auxtype of an existing file only if the file type is $06 (BIN).
- BASIC.SYSTEM can now launch (with "—") GS/OS applications. Files of type $B3 are passed through to an extended `QUIT` call to the ProDOS 8 MLI.
- $B3 files are now listed as `S16` in the catalog.
- Fixed a bug in the `READ` command where reading from the slot 3 `/RAM` disk passed errors back to BASIC, making the program break without completing a legal operation.
- Code optimized and crunched slightly.

Further Reference

- *BASIC Programming with ProDOS*
- *ProDOS 8 Technical Reference Manual*

ProDOS 8
#25: Non-Standard Storage Types

Revised by: Matt Deatherage December 1991
Written by: Matt Deatherage July 1989

This Technical Note discusses storage types for ProDOS files which are not documented in the *ProDOS 8 Technical Reference Manual.*

Warning: The information provided in this Note is for the use of disk utility programs which occasionally must manipulate non-standard files in unusual situations. ProDOS 8 programs should not create or otherwise manipulate files with non-standard storage types.

Changes since July 1989: Included new information on storing HFS Finder information in extended files' extended key blocks.

Introduction

One of the features of the ProDOS file system is its ability to let ProDOS 8 know when someone has put a file on the disk that ProDOS 8 can't access. A file not created by ProDOS 8 can be identified by the `storage_type` field. ProDOS 8 creates four different storage types: seedling files ($1), sapling files ($2), tree files ($3), and directory files ($D). ProDOS 8 also stores subdirectory headers as storage type $E and volume directory headers as storage type $F. These are all described in the *ProDOS 8 Technical Reference Manual.*

Other files may be placed on the disk, and ProDOS 8 can catalog them, rename them, and return file information about them. However, since it does not know how the information in the files is stored on the disk, it cannot perform normal file operations on these files, and it returns the `Unsupported Storage Type` error instead.

Apple reserves the right to define additional storage types for the extension of the ProDOS file system in the future. To date, two additional storage types have been defined. Storage type $4 indicates a Pascal area on a ProFile hard disk, and storage type $5 indicates a GS/OS extended file (data fork and resource fork) as created by the ProDOS FST.

Storage Type $4

Storage type $4 is used for Apple II Pascal areas on Profile hard disk drives. These files are created by the Apple Pascal ProFile Manager. Other programs should not create these files, as Apple II Pascal could freak out.

The Pascal Profile Manager (PPM) creates files which are internally divided into pseudo-volumes by Apple II Pascal. The files have the name PASCAL.AREA (name length of 10), with file type $EF. The `key_pointer` field of the directory entry points to the first block used by the file, which is the second to last block on the disk. As ProDOS stores files non-contiguously up from the bottom, PPM creates pseudo-volumes contiguously down from the end of the ProFile. `Blocks_used` is 2, and `header_pointer` is also 2. All other fields in the directory are set to 0. PPM looks for this entry (starting with the name PASCAL.AREA) to determine if a ProFile has been initialized for Pascal use.

The file entry for the Pascal area increments the number of files in the ProDOS directory and the `key_pointer` for the file points to `TOTAL_BLOCKS` - 2, or the second to last block on the disk. When PPM expands or contracts the Pascal area, `blocks_used` and `key_pointer` are updated accordingly. With any access to this entry (such as adding or deleting pseudo-volumes within PPM), the backup bit is not set (PPM provides a utility to back up the Pascal area).

The Pascal volume directory contains two separate contiguous data structures that specify the contents of the Pascal area on the Profile. The volume directory occupies two blocks to support 31 pseudo-volumes. It is found at the physical block specified in the ProDOS volume directory as the value of `key_pointer` (i.e., it occupies the first block in the area pointed to by this value).

The first portion of the volume directory is the actual directory for the pseudo-volumes. It is an array with the following Apple II Pascal declaration:

```
TYPE   RTYPE = (HEADER, REGULAR)

VAR    VDIR:  ARRAY [0..31] OF
                    PACKED RECORD
                           CASE RTYPE OF
                                  HEADER:       (PSEUDO_DEVICE_LENGTH:INTEGER;
                                                 CUR_NUM_VOLS:INTEGER;
                                                 PPM_NAME:STRING[3]);
                                  REGULAR:      (START:INTEGER;
                                                 DEFAULT_UNIT:0.255
                                                 FILLER:0..127
                                                 WP:BOOLEAN
                                                 OLDDRIVERADDR:INTEGER
                    END;
```

The `HEADER` specifies information about the Pascal area. It specifies the size in blocks in `PSEUDO_DEVICE_LENGTH`, the number of currently allocated volumes in `CUR_NUM_VOLS`, and a special validity check in `PPM_NAME`, which is the three-character string PPM. The header information is accessed via a reference to `VDIR[0]`. The `REGULAR` entry specifies information for each pseudo-volume. `START` is the starting block address for the pseudo-volume, and `LENGTH` is the length of the pseudo-volume in blocks. `DEFAULT_UNIT` specifies the default

Pascal unit number that this pseudo-volume should be assigned to upon booting the system. This value is set through the Volume Manager by either the user or an application program, and it remains valid if it is not released.

If the system is shut down, the pseudo-volume remains assigned and will be active once the system is rebooted. `WP` is a Boolean that specifies if the pseudo-volume is write-protected. `OLDDRIVERADDR` holds the address of this unit's (if assigned) previous driver address. It is used when normal floppy unit numbers are assigned to pseudo-volumes, so when released, the floppies can be reactivated. Each `REGULAR` entry is accessed via an index from 1 to 31. This index value is thus associated with a pseudo-volume. All references to pseudo-volumes in the Volume Manager are made with these indexes.

Immediately following the `VDIR` array is an array of description fields for each pseudo-volume:

```
VDESC: ARRAY [0..31] OF STRING[15]
```

The description field is used to differentiate pseudo-volumes with the same name. It is set when the pseudo-volume is created. This array is accessed with the same index as `VDIR`.

The volume directory does not maintain the names of the pseudo-volumes. These are found in the directories in each pseudo-volume. When the Volume Manager is activated, it reads each pseudo-volume directory to construct an array of the pseudo-volume names:

```
VNAMES: ARRAY [0..31] OF STRING[7]
```

Each pseudo-volume name is stored here so the Volume Manager can use it in its display of pseudo-volumes. The name is set when the pseudo-volume is created and can be changed by the Pascal Filer. The names in this array are accessed via the same index as `VDIR`. This array is set up when the Volume Manager is initialized and after there is a delete of a pseudo-volume. Creating a pseudo-volume will add to the array at the end.

Pascal Pseudo-Volume Format

Each Pascal pseudo-volume is a standard UCSD formatted volume. Blocks 0 and 1 are reserved for bootstrap loaders (which are irrelevant for pseudo-volumes). The directory for the volume is in blocks 2 through 5 of the pseudo-volume. When a pseudo-volume is created, the directory for that pseudo-volume is initialized with the following values:

```
dfirstblock = 0               first logical block of the volume
dlastblock = 6                first available block after the directory
dvid  = name of the volume used in create
deovblk = size of volume specified in create
dnumfiles = 0                 no files yet
dloadtime = set to current system date
dlastboot = 0
```

The *Apple II Pascal 1.3 Manual* contains the format for the UCSD directory. Files within this subdirectory are allocated via the standard Pascal I/O routines in a contiguous manner.

Storage Type $5

Storage type $5 is used by the ProDOS FST in GS/OS to store extended files. The key block of the file points to an extended key block entry. The extended key block entry contains mini-directory entries for both the data fork and resource fork of the file. The mini-entry for the data fork is at offset +000 of the extended key block, and the mini-entry for the resource fork is at offset +$100 (+256 decimal).

The format for mini-entries is as follows:

storage_type	(+000)	**Byte**	The standard ProDOS storage type for this fork of the file. Note that for regular directory entries, the storage type is the high nibble of a byte that contains the length of the filename as the low nibble. In mini-entries, the high nibble is reserved and must be zero, and the storage type is contained in the low nibble.
key_block	(+001)	**Word**	The block number of the key block of this fork. This value and the value of storage_type combine to determine how to find the data in the file, as documented in the *ProDOS 8 Technical Reference Manual*.
blocks_used	(+003)	**Word**	The number of blocks used by this fork of the file.
EOF	(+005)	**3 Bytes**	Three-byte value (least significant byte stored first) representing the end-of-file value for this fork of the file.

Immediately following the mini-entry for the data fork may be up to two eighteen-byte entries, each with part of the HFS Finder information for this file. The first entry stores the first 16 bytes of the Finder information, and the second entry stores the second 16 bytes. The format is as follows:

entry_size	(+008)	**Byte**	Size of this entry; must be 18 ($12).
entry_type	(+009)	**Byte**	Type of this entry—1 for FInfo (first 16 bytes of Finder information), 2 for xFInfo (second 16 bytes).
FInfo	(+010)	**16 Bytes**	First sixteen bytes of Finder Info.
entry_size	(+026)	**Byte**	Size of this entry; must be 18 ($12).
entry_type	(+027)	**Byte**	Type of this entry—1 for FInfo (first 16 bytes of Finder information), 2 for xFInfo (second 16 bytes).
xFInfo	(+028)	**16 Bytes**	Second sixteen bytes of Finder Info.

Note: Although the ProDOS FST under GS/OS will only create both of the mini-entries, as described above, the ProDOS File System Manager (ProDOS FSM) for the Macintosh, which is part of the Apple IIe Card v2.0 software, may create only **one** of the entries, so you may find an `entry_type` of 2 at offset +009 in the block. If one of the entries is missing, it should be considered to be all zeroes.

All remaining bytes in the extended key block are reserved and **must** be zero.

Further Reference

- *Apple II Pascal ProFile Manager Manual*

- *GS/OS Reference*
- *ProDOS 8 Technical Reference Manual*

ProDOS 8
#26: Polite Use of Auxiliary Memory

Written by: Matt "Missed Manners" Deatherage January 1990

This Technical Note discusses the use of auxiliary memory, particularly the reserved areas, and this information supersedes the discussion in the *ProDOS 8 Technical Reference Manual.*

"I want to use auxiliary memory!"

Dear Missed Manners:

I'm having difficulty in a program I'm writing for 128K Apple II computers. My program is about to run out of memory. I have squeezed, packed and compressed this program until I can simply cajole no more room from it, and yet more room it needs. Apple has a large section of memory reserved, but my investigations reveal that this memory (in a language card, where it is doubly valuable since it stays put when main memory is swapped) seems to be unused. The *ProDOS 8 Technical Reference Manual* states unfailingly that the memory must not be used, but it seems to be wasting away! How can I politely use this valuable resource in my own application?

Gentle Developer:

Polite programming requires cooperation by both developers and system software, and it is the users who suffer when that cooperation is not maintained. Apple reserves memory for system software so that it can expand without breaking applications. Missed Manners hopes that he is not being too presumptuous by assuming that you would be appalled if Apple was required to expand ProDOS 8 and reclaim the memory from $B000 through $BFFF. He notes this situation would not be necessary if Apple were able to use memory it currently has reserved for such purposes.

However, if necessity requires more memory for your application, a polite inquiry to Apple may be sent. "Would it be possible for me to use some of Apple's reserved memory in my application without compatibility problems?" would be a polite request, for example. Using the memory without asking or demanding action would not only be impolite, it would pose future problems for an application. Those who do not program politely will eventually regret such a decision.

Conflicts and Arbitration

Some of the polite letters Apple has received on this subject point out that the built-in /RAM device uses almost all of the memory marked as “reserved” in the ProDOS 8 memory map. How can the system software expand into areas it’s already using?

It can't, of course…unless it already has and you don't know it. This is partially the case. On the Apple IIGS, memory can be obtained through the Memory Manager, so adding new components to the system software is relatively easy. If memory is available, it is allocated by the Memory Manager and used by the application. If memory is not available, the program trying to install the component in question is told and the component is not installed. (If a vital part of the system can't be installed, the boot process grinds to an unceremonious, but grammatically correct, halt.)

Since the 8-bit Apple II family has no memory manager, applications and system software must mutually (and politely) agree which areas of memory belong to whom. If the system software is broken into components, some memory will be reserved for components which are not present at a given time. This is largely the case with the auxiliary language card memory on the 128K Apple II.

The area from $D100 through $DFFF in bank 2 of the auxiliary language card is for the use of third-party RAM-based drivers, to be discussed in a future ProDOS 8 Technical Note. At least one version of Apple II SANE is configured to load at $E000 in the auxiliary language card, which is perfectly acceptable since SANE is part of the system software (it just doesn't ship with the system).

Clearly, /RAM can't use this memory at the same time the system software does. This very dichotomy gives the Rule of Auxiliary Memory that simplifies this memory management.

> **The Rule of Auxiliary Memory:** If /RAM is enabled, all auxiliary memory above location $800 may be used by an application after first removing /RAM as discussed in the *ProDOS 8 Technical Reference Manual*. /RAM should be reinstalled upon completion.
>
> If /RAM is not enabled, then auxiliary memory above $800 may be used at the application programmer's discretion, but the areas marked as reserved must be respected.

System software use of this area should be denoted by the absence of /RAM. This means that if ProDOS 8 were to ever expand to run only on 128K machines and *require* auxiliary language card memory, that no /RAM device would be installed by default. Although this seems unlikely, it is nonetheless another indicator that your application should not depend on /RAM to operate.

Similarly, if /RAM is not present when your application is launched, you may **not** reenable it. If it is present, you may remove it to use the memory if you reinstall it when you're done.

Also note that auxiliary memory below $800 that is not on the 80-column text screen is always reserved and may never be used by applications.

Applications which use reserved memory areas without observing this rule run the risk of storing data over third-party RAM-based drivers (rendering their software useless to peripherals that may require such drivers, like third-party networks, devices for the visually impaired, or closed-system hard disks) or future system software.

Further Reference

- *ProDOS 8 Technical Reference Manual*

Apple II
Technical Notes

Developer Technical Support

ProDOS 8
#27: Hybrid Applications

Written by: Matt Deatherage March 1990

This Technical Note discusses considerations for "hybrid" applications, which use Apple IIGS-specific features from ProDOS 8.

Why Use Hybrid Features?

There are many reasons not to write hybrid applications. If your target machine is the Apple IIGS, it's pretty silly to write a ProDOS 8-based application. You are limited to the slower I/O model of ProDOS 8, you cannot access foreign file systems or large CD-ROM volumes, you cannot reliably access the toolbox (patches to the toolbox are only loaded when GS/OS is booted, which forces you to require GS/OS to be booted), and you cannot work with desk accessories that do disk access (CDAs cannot reliably "save and restore" an area of bank zero to use for ProDOS 8 disk access because they don't know if an interrupt handling routine is located there).

However, applications targeted for all Apple II computers may reasonably wish to take advantage of IIGS features. For example, a word processor or telecommunications program may want to use extra IIGS memory. This Note is your spiritual guide to such features.

Memory Management

Applications wishing to use extended (beyond the lower 128K) memory on the IIGS must, like all IIGS applications, get it from the Memory Manager. This is not a consideration for non-hybrid applications for two reasons. First, when GS/OS launches a ProDOS 8 program, it reserves all of the lower 128K memory for ProDOS 8, so no other component (tool, desk accessory, INIT) can accidentally use that memory. (In fact, if some of the memory is not available, GS/OS refuses to launch ProDOS 8 at all.) Second, when ProDOS 8 is directly booted, none of the memory is allocated since these other components, which might be using the Memory Manager, aren't loaded either.

If your ProDOS 8 application was launched by GS/OS, all of the managed lower 128K has already been allocated for you by GS/OS. If you call `MMStartUp`, the user ID returned is one belonging to GS/OS. In such cases, the auxiliary field of the user ID is already being used by GS/OS and must **not** be altered by your application. You also must not call any Memory Manager routine which works on all handles of a given user ID, such as `DisposeAll` or

`HUnlockAll`. You must manage all handles individually and not by user ID. You may, if you wish, call `GetNewID` to get a new user ID for use in a user ID-based memory management system. The ID should be of type $1000 (application).

You can tell whether your application was launched by GS/OS by checking `OS_BOOT`, the byte value at $E100BD. `OS_BOOT` is $00 when the boot OS was ProDOS 8, indicating that your application was not loaded by GS/OS. If this is the case and you want to use extended IIGS memory, you should call `GetNewID` to obtain a new application ID then use `NewHandle` to allocate four handles to hold the memory normally reserved for ProDOS 8 by GS/OS. You should obtain memory at $00/0800 (size $B800), $01/0800 (size $B800), $E0/2000 (size $4000) and $E1/2000 (size $8000). You may then use `MMStartUp` to register yourself with the Memory Manager; `MMStartUp` fails if it's being called from an unallocated memory block, so you must allocate the memory your application occupies first.

Further Reference

- Apple IIGS Technical Note #17, Application Startup and the MMStartUp User ID

Apple II
Technical Notes

Developer Technical Support

ProDOS 8
#28: ProDOS Dates—2000 and Beyond

Written by: Dave Lyons September 1990

This Technical Note explains how ProDOS year values range from zero to ninety-nine and represent the years 1940 through 2039.

The ProDOS date format uses sixteen bits: seven bits for the year, four for the month, and five for the day (see the *ProDOS 8 Technical Reference Manual*, page 71). Dates are represented in this format in the parameter blocks for ProDOS 8 MLI calls and in the directories of ProDOS volumes.

In seven bits, 128 different years could be represented, but the proper interpretation of those bits has never been defined clearly until now.

2000? I'll Be Dead By Then Anyway

It's only nine years, folks, and then things get weird. Is that ProDOS year 100 or ProDOS year 0? How do you compare two file-modification dates so it keeps working correctly?

Before you dismiss questions like this, consider just how sure you are that nobody will be using your software in nine years, or whether those few dedicated weirdos are going to call you up on January 1, 2000 to complain. There will be **plenty** of computer-related problems in 2000, so write your applications right today.

Some Choices

These two possible interpretations were considered and then rejected in favor of The Definition below.

1. Valid years would be from 0 to 99, meaning 1900 to 1999, so ProDOS dates would just "expire" at the end of 1999. No fun.

2. Valid years would be from 0 to 127, meaning 1900 to 2027. This is a little better, except that almost no existing software is prepared to deal with year values outside the 0-to-99 range.

So, you are left with...

The Definition

The following definition allows the same range of years that the Apple IIGS Control Panel CDA currently does:

- A seven-bit ProDOS year value is in the range 0 to 99 (100 through 127 are invalid)
- Year values from 40 to 99 represent 1940 through 1999
- Year values from 0 to 39 represent 2000 through 2039

Note: Apple II and Apple IIGS System Software does not currently reflect this definition.

How to Compare Two Years

To compare two dates, you need to adjust the years to allow for the wrap-around effect between 39 and 40. A simple approach is to add 100 to any year less than 40 before doing the comparison, thus comparing two values in the range 40 to 139.

```
    CompareAB    lda YearB
                 cmp #40
                 bcs B_OK
                 adc #100  ;carry is clear
                 sta YearB

B_OK             lda YearA
                 cmp #40
                 bcs A_OK
                 adc #100  ;carry is clear
                 sta YearA

A_OK             cmp YearB
                 bcc A_is_earlier
                 ...
```

What About GS/OS Dates?

This definition affects how the GS/OS ProDOS File System Translator works internally, but it does not affect GS/OS applications. A year value under GS/OS is always a byte offset from 1900, giving a possible range of 1900 to 2155, regardless of the file system involved.

What Do You Do After 2039?

Apple is still working on it. Contact your neighborhood Apple Developer Technical Support office in 2030.

Further Reference

- *ProDOS 8 Technical Reference Manual*
- *Apple IIGS Toolbox Reference Manual,* Volume 1
- *GS/OS Reference*

ProDOS 8
#29: Clearing the Backup Needed Bit

Written by: Jim Luther September 1990

This Technical Note shows how to clear the "backup needed bit" in a directory entry's `access` byte.

If you are writing a file backup utility program, you probably want to clear the backup needed bit in each directory entry's `access` byte as you make the backup of the file associated with that directory entry. The `SET_FILE_INFO` MLI call normally sets the backup needed bit of the `access` byte, but how do you clear it? The answer is at location `BUBIT` ($BF95) on the ProDOS 8 system global page.

`BUBIT` normally contains the value $00. When `BUBIT` contains $00, the `SET_FILE_INFO` MLI call **always** sets the backup needed bit in the directory entry's `access` byte. However, if the value $20 is stored in `BUBIT` immediately before calling `SET_FILE_INFO`, the backup needed bit in the directory entry's `access` byte can be cleared. `BUBIT` is set back to $00 by the MLI call. The following code example shows how to clear the backup needed bit. Values other than $20 or $00 in `BUBIT` are not supported.

```
; The pathname of the file should be in ThePathname buffer when this code is called!

               65816 off
               longa off
               longi off

ClearBackupBit start

; System global page locations

MLI            equ $BF00               ;MLI call entry point
BUBIT          equ $BF95               ;Backup Bit Disable, SET_FILE_INFO only

; MLI call numbers

SET_FILE_INFO  equ $C3
GET_FILE_INFO  equ $C4

; set up FileInfoParms for GET_FILE_INFO MLI call
               lda #$0A
               sta param_count
; then...
               jsr MLI                 ;get the current file info
               dc  I1'GET_FILE_INFO'
               dc  I2'FileInfoParms'
               bne Error
```

```
               lda #$20                ;set the backup bit disable bit
               sta BUBIT
               eor #$FF
               and access              ;clear the backup needed bit
               sta access

; set up FileInfoParms for SET_FILE_INFO MLI call
               lda #$07
               sta param_count
; then...
               jsr MLI                 ;set the file info with the file info
               dc  I1'SET_FILE_INFO'   ;(clearing only the backup needed bit)
               dc  I2'FileInfoParms'
               bne Error
               rts                     ;return to caller

Error          anop                    ;routine to handle MLI errors
               rts

; Parameter block used for GET_FILE_INFO and SET_FILE_INFO MLI calls

FileInfoParms  anop
param_count    ds  1
pathname       dc  i2'ThePathname'
access         ds  1
file_type      ds  1
aux_type       ds  2
storage_type   ds  1
blocks_used    ds  2
mod_date       ds  2
mod_time       ds  2
create_date    ds  2
create_time    ds  2

ThePathname    entry
               ds  65                  ;store the pathname of the file here

               end
```

Further Reference

- *ProDOS 8 Technical Reference Manual*

Apple II
Technical Notes

Developer Technical Support

ProDOS 8
#30: Sparse Station

Written by: Matt Deatherage May 1992

This Technical Note discusses issues when using sparse files under ProDOS 8.

Sparse Information Available

The concept of sparse files is introduced in the *ProDOS 8 Technical Reference Manual* in sometimes confusing language. The concept behind sparse files is pretty simple. If you didn't think it could be explained in two paragraphs, have a seat and learn something.

The ProDOS file system keeps track of where files reside on disk through a series of "index blocks." All index blocks are disk blocks that contain lists of block numbers. They may be organized in several ways (seedling, sapling or tree), depending on how big the file is—one 512-byte block can hold 256 two-byte block numbers. If a file is one block long, it has no index blocks and is a **seedling** file. If a non-sparse file is between two and 256 blocks long, it has one index block and is a **sapling** file. If a non-sparse file is longer than 256 blocks, it's a **tree** file and has a "master index block" that points to other index blocks. This is more than enough to store any ProDOS file—one master index block pointing to 256 other index blocks, each of which points to 256 data blocks on disk would be a 32 MB file—twice the limit of 16 MB imposed by ProDOS's 3-byte storage for file lengths.

What happens if you don't need to use all of those blocks? For example, if you need to store data at file offset $0000 and at file offset $20000, does ProDOS make you waste 256 disk blocks you're not going to use? Fortunately, the answer is "no." ProDOS lets you skip any data block you're not using by recording a pointer to data block $0000 instead of to a regular block on the disk. When ProDOS sees a block pointer of $0000 in an index block, it knows not to read block zero (which contains boot code) but instead to pretend that it read a block of zeroes from the disk. This lets you save lots of space on disk—a file created this way is a **sparse** file. (See? Two paragraphs.)

Under ProDOS 8, you can create a sparse file by using the `SET_EOF` MLI command to extend the file's current end-of-file position, and then using `SET_MARK` to move the mark to the new end-of-file position. If you grow a file by increasing the EOF but not actually writing data, ProDOS 8 makes the blocks you skip sparse. Under GS/OS, the ProDOS FST automatically converts long stretches of zeroes to sparse blocks, making sparse files even more prevalent.

Dealing With Sparsity

Unfortunately, ProDOS 8 does **not** automatically make sparse files when you write large sections of zeroes. That means if you read a sparse file and write it back out, you "expand" it and it's no longer sparse. The file could balloon to hundreds of times its previous disk space, which is not a good thing.

So how do you recognize a sparse file? You can notice that the length of the file has to be pretty close to 512 bytes multiplied by the number of blocks allocated to data in the file. For example, take a file that's $4068 bytes long. $4068 bytes takes 33 512-byte blocks—32 blocks is $4000 bytes, plus one more block for the last $68 bytes. This is between 2 and 256 blocks, so there's one more block allocated for the index block. If this file is not sparse, it uses 34 blocks on disk. If it uses any **less** than 34 blocks in reality, it's sparse.

This calculation gets a little tricker for tree files—if the file has more than 256 data blocks, add one master index block plus one index block for each 256 data blocks or portion thereof. To give another example, a file that's $68D3F bytes long takes 839 ($347) data blocks. This file has five additional blocks allocated to it—one master index block and four index blocks. The first three index blocks are full (256 × 3 = 768) and the fourth contains the remaining 71 data blocks. If this file takes less than 844 blocks on disk, it's sparse.

Too Complicated?

For all except very speedy utilities to copy files, yes. If you just need an easy way to deal with sparse files that's not so speed-critical, read on.

All you have to do to preserve (or create) sparsity in normal file copying operations is scan the data you've read from disk before you write it back. Suppose your file copying buffer is 10K large. Read 10K of data from your source file, then divide the buffer into 512-byte chunks and scan the data looking for zeroes. If you find a non-zero byte, write the entire 512-byte chunk of data to the target file and proceed to the next 512-byte chunk. If you don't find any non-zero bytes in a 512-byte chunk, just set the mark ahead 512 bytes and don't issue a `WRITE` call. This is basically how GS/OS's ProDOS FST automatically sparses files, and it can work for you too.

Is It That Easy?

Well, no. There's an important exception—AppleShare.

Most AppleShare servers (including all of Apple's) don't support sparse files—all the logical blocks you use have to be physically allocated on the server's hard disk. The following BASIC.SYSTEM command:

```
BSAVE SPARSE.FILE,A$300,L$1,B$FFFFFF
```

creates a 16 MB sparse file with one byte of logical data in it. This file only takes 5 blocks on a ProDOS disk (one master index block, two index blocks and two data blocks—it takes two data blocks because ProDOS 8 always allocates the very first block of a file when you create it, even if you don't use the first 512 bytes), but it takes 16 MB of disk space on a server.

That's not all—for speed reasons, AppleShare does **not** fill the extra, normally-sparsed blocks with zeroes. If you issued the above command to an AppleShare server under ProDOS 8 and then tried to read the first few bytes of the resulting file, they would be garbage—but not zeroes.

ProDOS 8 Technical Note #21 gives information on identifying AppleShare server volumes—if you're dealing with one, do **not** use normal sparse file creation techniques. Just write the 512 bytes of zeroes instead of advancing the mark. It doesn't take any more disk space and it achieves the results you want.

One More Thing

In versions of ProDOS 8 up to 1.9, setting the end-of-file position past $200 on a **seedling** file created a sparse file that confused ProDOS 8 if you ever used `SET_EOF` on it again. This is fixed in version 2.0.1 and later.

Further Reference

- *ProDOS 8 Technical Reference Manual*