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