NOTE: This Technical Note has been retired. Please see the Technical Notes page for current documentation.

Technical Note DV21

Serial PollProc

CONTENTS

This Technical Note discusses how to make a `PollProc` for your MIDI (Musical Instrument Digital Interface) driver on the Macintosh PowerBook 140 and 170.

[Jun 01 1992]

For MIDI Consumption Only

You are writing your own MIDI driver and your driver does not fully work on the PowerBook 170/140. The PollProc support that might help solve your problem has been undocumented until now because it has a bug in it which if ever fixed would cause major problems with every PollProc ever made. The bug is in the way that the PollProc mechanism handles data errors - it doesn't. At some point this might get fixed and an fix would require changes to any existing PollProcs. We are only documenting this now because we (Apple) would like to see high speed MIDI data transfers working on the PowerBook 170/140, and the PollProc support is the only solution we have been able to come up with. If you do use this information, be aware that the PollProc mechanism may change in the future and when it does your PollProc will need to change. We do not recommend the use of PollProc mechanisms on any other Macintosh computers.

Back to top

What the Problem Is

When doing a large data dump, such as downloading MIDI instruments or sampled sounds, MIDI data overruns the input serial port on the PowerBook 170/140.

Background

- MIDI developers and users have been reporting problems that occur on PowerBook 170/140. The specific problem described by the developers is that data overrun errors occur (that is, serial data is lost). MIDI data is serial data that is transmitted at 31.25 Kbaud. This means that one byte of data is transmitted approximately every 200 usec.
- The serial port has a three-byte FIFO, which means that three bytes of data could be stored temporarily before a data overrun (data loss).
- The MIDI functioned OK on the original portable, but just barely. The overhead of communicating to the power

manager microprocessor seems to interfere with MIDI.

- The 170/140 hardware required certain changes to the power management software because of changes in the hardware. In particular, the hardware changes required changes to the protocol used to communicate to the power manager microprocessor.
- The 170/140 has software backlight controls that cause constant communication between the 68030 and the power manager microprocessor (every 200 msec).

Findings

- The MIDI driver loses data. The 170/140 has a real-time problem and is not able to keep up with sustained MIDI data rates. The culprit is the communication between the 68030 and the power manager microprocessor. During this communication, interrupts must be disabled for a certain amount of time.
- On the 170/140, the protocol for this communication was changed from that of the portable. The 170/140 can cause interrupt blackouts up to 6 msec as compared to approximately 500-700 usec on the portable (estimation only). Assuming the worst case, during the 6 msec blackout as many as 30 MIDI data bytes could have been sent. Since the FIFO on the serial port is only 3 deep, this means that as many as 27 bytes could have been lost (remember these are ballpark figures only).
- The problem is aggravated by increased power manager communications for backlight controls.

Back to top

What the Solution Is

- Changing the protocol to the power manager microprocessor (given the hardware constraints) is not practical since the problem is not completely solved and could cause other system problems.
- At the moment, no Apple-only solution is possible.
- A developer-only solution is possible. Currently an internal mechanism exists to keep up with high data rates on the modem port. This mechanism, called PollProc (Polling Procedure), will allow the ROM code to handle the serial port during known interrupt blackout windows, which helps prevent data loss. The power manager communication software currently checks for such a routine and will use it automatically if it is present. In addition to correcting this problem, this will also allow MIDI to perform during floppy activity (which has similar real-time problems) since the floppy driver also checks for PollProc.

In the code which is included at the end of this Technote, there is a extra Procedure which is call `ProcessByte`. In the sample this routine does nothing. The reason for the sample not doing anything is due to the nature of the routine. What the routine does is completely dependent on what the serial driver is doing or wants to do with the data as it is read into the machine. This routine might be used to decompress data, compress data, decoded the data, or do any other kind of alteration you wish to do to it. The Macintosh OS does not do anything to the data, so this routine is not needed, but your application might need this routine - it is up to you, just don't do to much at this time. It is important to remember that you need to get in and out of the PollProc as fast as possible.

Back to top

What Is a PollProc and How Does It Work?

A PollProc is a routine that a serial driver implements so it can still get data when the OS turns interrupts off for a significant amount of time. Although PollProc mechanisms work for generic serial drivers, it is recommended that you use this feature in your MIDI driver only on the PowerBook 170/140. When the MIDI driver is opened and supports PollProc mechanisms, it needs to place a pointer to this routine in the low-memory global - PollProc. When the OS (such as the Power Manager and floppy driver) turns off interrupts, it checks to see if the low-memory global is nil or not. If the global is not nil, then it the OS will poll the SCC for incoming data and stuff the data into a buffer. Then just before the OS turns the interrupts back on, it calls the PollProc and passes the buffer to it. The PollProc will be able to handle the data as if it were coming in via the serial port.

The PollProc is supported only on port A. Port B PollProcs are not supported.

The comments in the following code give more detail about how to implement the PollProc.

Back to top

PollProc Sample Code

```
;_____________________________________________________________________________
;InputPollData  -  process SCC input data
;
;This routine is called via the low-memory vector PollProc by system code
;that had interrupts disabled for a long enough period of time that SCC
;data may have been lost. The system code will poll the SCC for data during
;the time it had interrupts disabled and call this routine right before
;interrupts are reenabled. The address of the InputPollData routine should
;be written into the "PollProc" low-memory vector when the SCC channel A
;driver is opened. The "PollProc" low-memory vector should be zeroed
;when the driver is closed.
;
;The InputPollData routine will be called with data to be processed on the
;stack. This routine should process the data as if it had been received by
;the driver's receive data interrupt routine.
;
;Note:     PollProc mechanisms are not necessary on SCC IOP based machines and
;should not be used.
;
;Input:    a6.l  =  SCC channel A data pointer
;Output:    none
;
;allowed to trash:    a0-a1/a3-a4

PollStack     equ     $13A      ; SCC poll data start stack location [pointer]
PollProc      equ     $13E      ; SCC poll data procedure [pointer]
RxCA        equ    0    ; Bit zero of SCC RR0 indicates receive char avail.

InputPollData

movea.l    (sp)+,a4        ; Save return address.
movea.l    PollStack,a3        ; a3 = ptr to beginning of data on stack.

;First empty all the data from the SCC. This may not be needed, but it is
;here for completeness. The drivers that will use the PollProc mechanism
;will already have similar code to this, so whether you implement this or
;not is more of a personal call. Our recommendation is that you try to go
;without the code, and if you find you do need it, then implement it.

@EmptySCC
movea.l    SCCRd,a0        ; base addr of SCC read register 0 from low mem
addq.w        #2,a0            ; Add offset to get to channel A registers.
btst.b        #RxCA,(a0)        ; Test if SCC data is available.
beq.s        @ProcessData        ; no additional SCC data
move.b        (a6),-(sp)        ; Move SCC channel A data onto stack.
bra.s        @EmptySCC

;Process all the SCC data on the stack as if it were read in normally by
;the SCC driver's receive interrupt routine. There is stack data starting
```

```
;from the address in the low-mem PollStack, to the current stack pointer.

@ProcessData
cmp.l        sp,a3        ; Have we processed all the stack data?
beq.s        @Done        ; We are done.
subq.w        #1,a3        ; Skip over garbage byte because stack pushes words.
move.b        (a3)+,d0    ; Get the saved data byte.

bsr.s        ProcessByte    ; Call driver routine to process the data byte.
bra.s        @EmptySCC    ; Check for SCC data before processing next saved byte.

;Done - cleanup stack of saved data

@Done
move.l        PollStack,sp        ; Set stack ptr to pop saved data.
jmp        (a4)            ; Jump to the return address.

;_____________________________________________________________________
;ProcessByte  -  process saved SCC input data
;
;This routine is a stub example routine that will process a saved data
;byte as if the driver had read in the byte normally.
;
;Input:    d0.b  =  SCC channel A data byte
;Output:    none
;

ProcessByte

;Fill in necessary code.

rts
```

Back to top

Downloadables

Acrobat version of this Note (K) Download

Back to top
