ATN Test

by Troy Fullwood

Introduction

The Commodore 64 implements a single-tasking operating system called the `KERNAL` – legendarily due to the designer misspelling the word in their notebooks and this getting copied verbatim into the documentation. By single-tasking OS, I mean a Hardware Interface Layer, which is all that really means. A multitasking operating system also provides a resource multiplexor; many different processes competing for the screen, RAM, hard drive, keyboard, etc etc.

As a single-tasking operating system, none of the I/O routines provided by the `KERNAL` are asynchrynous; they all "block" until the call is completed. In the case of Commodore programs that, for instance, play music while loading, they often use a custom fast-loader and add interrupts that will call the music in the background. Even the C64 version of the Geck/OS system – which supports multitasking – will pause whenever accessing the disk drive.

The disk drive – as well as most other peripherals not on the user port – is connected daisy-chain style via the Commodore Serial Bus. The Serial protocol allows for, at any time, one talker and one or more listeners. First, the talker advertises it has a byte to send,. The listeners all acknowledge they're ready to receive – taking as long as they want to do so. This is so that a printer could have time to empty its buffer, or a disk drive have time to flush to disk. Then, the talker acknowledges – and we're now LOCKED IN to timing critical code to transfer 8 bits. And the process begins again.

If the talker takes more than 256 microseconds to respond when the listeners are ready, then this byte is the last of the stream, following which is end of file. If the talker times out at more than 512 microseconds to respond, then the stream is assumed to be empty. A reasonable response to a device not being on the lin

Unfortunately for us, this timing setup means that it's not possible to use the Commodore Serial Bus as a non-blocking I/O system.

Except, for the ATN line.

Only the computer has control over the ATN line. Whenever it activates it – *even mid byte transfer* – all devices on the bus are supposed to respond within 1000 microseconds. Now, the computer is the talker, and will transfer bytes as long as the ATN line is held down. These bytes are sent as normal, with the computer indicating it has a byte to send, listeners taking as long as they please to reply OK, and the talker then sending the byte.

The question is, how will a device on the receiving end of an ATN command respond to being left on the hook for, say, several seconds. They're supposed to ignore the 256-512 microsecond "end of stream" delay indicator, since the command end is indicated by dropping the ATN line. Glancing at the 1541 drive ROM source, it appears to me that the system will completely ignore any such delays, no matter how long.

If serial bus devices *do* completely ignore the delay, I believe I could implement a non-blocking asynchronous serial bus library. I could have a routine called by a regular interrupt, such as the system clock. I could transfer a few bytes, using the VIC-II scanline counter as a way to tell how long I've been transferring. Once I'm out of itme, I could send an ATN command to pause the stream. If they took too long to respond to the ATN command, I just hold them on the line and not send the byte til the next interrupt.

I want to test this, both with a disk drive on serial bus 8, and a printer on serial bus 4.

ATN Commands

There are several kinds of ATN commands which may be sent.

- \$20-\$3E: `LISTEN`. Commands the device numbered in the lower five bits to become a listener.
- \$3F: `UNLISTEN`. Commands all listeners to stop listening.
- \$40-\$5E: `TALK`. Commands the device numbered in the lower five bits to become a talker. Then, the computer could swap over to a listener after the ATN stream. This allows, for instance, a disk file to be `TALK`'ed by the 1541, and the computer to `LISTEN` it into it's RAM.
- \$5F: `UNTALK`. All talkers are commanded to shush up.
- \$60-\$7E: `SECOND`. Send the secondary address in the lower 5 bits. Can optionally be send after a `TALK`/`LISTEN`.
- $E0-EF$: `CLOSE`. Prefix this with a `LISTEN` and follow it with an `UNLISTEN` in the same ATN command. This will command a file associated with the secondary address in the lower 4 bits to be closed.
- $F0-FF$: `OPEN`. Prefix this with a `LISTEN` in the same ATN command. Associate a named file with the secondary address in the lower 4 bits. The filename is sent as part of the ATN command after the `OPEN`, followed by an `UNLISTEN`.

The Test

Let's write our own set of serial routines. Let's set-up our high level interrupt routine. It'll get called 60 times per second according to the clock interrupt set up when the C64 boots into BASIC. The first thing it does is save the current and target scanlines, so we can use the VIC-II's scanline counter to know when to wrap up our transfer.

«irq»=

```
.code
.export MyIrq
```

```
.proc MyIrq
    lda VIC_SCANLINE
    sta start_line
    clc
    adc #LINES_FOR_XFER
    sta target_line

<<transfer_bytes>>

<<wrapup_irq>>
.endproc
```

We need to define the scanline register.

«constants»=

```
VIC_SCANLINE = $D012
```

And the number of lines we intend to use this go-round.

«constants»+

```
CYCLES_PER_LINE=65 ; on NTSC, PAL is 63
TARGET_MICROSECONDS = 10000
LINES_FOR_XFER = TARGET_MICROSECONDS / CYCLES_PER_LINE
```

Plus those line variables.

«variables»=

```
    .bss
start_line:  .res 2
target_line: .res 2
```

The Transfer

To transfer the bytes, we need to know what bytes to transfer. We either want to read n bytes from the device, write n bytes to the device, or send an ATN command. The ATN command preempts the actual transfer, since the transfer itself might send an ATN command to pause.

«constants»+

```
READ = $FF
WRITE = $00
```

«variables»+

```
    .bss
atn_bytes: .res 1 ; indicates the number of bytes we want to transfer
    .export atn_bytes
atn_index: .res 1 ; indicates the current index into the ATN transfer
```

```
    .export atn_index
atn_buffer: .res MAX_ATN_BYTES ; holds the actual ATN bytes to send
    .export atn_buffer
ser_rw: .res 1 ; indicates a read or a write
    .export ser_rw
ser_bytes: .res 1 ; indicates the number of bytes to read/write over serial
    .export ser_bytes
ser_index: .res 1 ; indicates the current index into the serial transfer
    .export ser_index
    .zeropage
ser_pointer: .res 2 ; pointer to wherever we want to send/receive data
    .export ser_pointer
ser_eof: .res 1 ; flag for the end of the current transfer is also end of file
```

«constants»+

```
MAX_ATN_BYTES = $FF
```

Now, let's transfer some bytes!

«transfer_bytes»=

```
XferLoop:
    lda atn_bytes   ; check if we have any ATN command bytes we wanna send
    bne SendAtn
    lda ser_bytes   ; check if we have any normal data bytes we wanna send
    bne :+
    jmp XferDone
:
<<read_or_write>>
    jmp XferLoop
SendAtn:
<<send_atn>>
```

Sending ATN's Let's get into the ATN stuff, since that's what we care about.

First, let's make sure the ATN line is actually on.

«send_atn»=

```
    bit atn_on_flag
    bmi SkipTurnOn
    jsr AtnOn
    jsr ClkOn
    jsr DataOff
    jsr Wait1kUs
    lda #$FF
    sta atn_on_flag
SkipTurnOn:
```

«variables»+

```
    .bss
atn_on_flag: .res 1
    .export atn_on_flag
```

Now, we grab the byte and try to send.

«send_atn»+

```
    ldy atn_index
    lda atn_buffer,y
    sta byte_buffer
    jsr TryWrite
```

«variables»+

```
    .bss
byte_buffer: .res 1
    .export byte_buffer
```

TryWrite returns carry set if succeeded, carry clear if failed. Failure would be due to a timeout. If we timed out, just end the transfer interrupt. We'll try to send the ATN byte again next go-round, and the ATN line will be kept on in the meantime, forcing the other devices to keep waiting for us to send it.

«send_atn»+

```
    bcc XferDone
```

On the other hand, if we succeeded, increment our buffer index. If we wrote all the bytes, then end the ATN command.

«send_atn»+

```
    ldx atn_index
    inx
    cpx atn_bytes
    beq AtnDone
    stx atn_index
    jmp XferLoop
AtnDone:
    jsr AtnOff
    lda #0
    sta atn_bytes
    sta atn_index
    sta atn_on_flag
    jmp XferDone
```

Regular Serial Transfer

Serial Writes We check if we're doing a read or a write.

«read_or_write»=

```
    .assert READ>=$80 && WRITE<$80,error,"expect to BIT a r/w flag"
    bit ser_rw
    bmi SerRead
SerWrite:
<<ser_write>>
SerRead:
<<ser_read>>
```

To start our write, we must output a LISTEN ATN command to the other device, so we can TALK to it.

«ser_write»=

```
    bit ser_online_flag
    bmi WriteOnline
    lda #LISTEN
    clc
    adc ser_dev
    sta atn_buffer
    lda #SECOND
    clc
    adc ser_second
    sta atn_buffer+1
    lda #2
    sta atn_bytes
    lda #0
    sta atn_index
    lda #$FF
    sta ser_online_flag
    bne XferLoop            ; Always taken. This will proceed with sending the ATN LISTEN co
WriteOnline:
```

«variables»+

```
    .bss
ser_online_flag: .res 1
    .export ser_online_flag
ser_dev: .res 1
    .export ser_dev
ser_second: .res 1
    .export ser_second
```

«constants»+

```
LISTEN = $20
SECOND = $60
```

Once *ser_online_flag* is set, the target device is listening. Even if we ended the interrupt routine in the middle of sending the LISTEN command, the device kept waiting for the ATN to finish for as long as we needed, and in that case it would be listening in now on this round through the interrupt.

Let's grab that byte and try to send it!

«ser_write»+

```
    ldy ser_index
    lda (ser_pointer),y
    sta byte_buffer
    jsr TryWrite
```

As for the ATN command, TryWrite returns carry clear if there was a timeout, carry set if we succeeded.

If we timed out, then we want to send an ATN command to have the device stop listening. This ATN command will also time out, which is fine.

«ser_write»+

```
    bcs GoodWrite
    lda #UNLISTEN
    jsr AtnOne
    jmp XferLoop
```

«subrs»=

```
    .code
    .export AtnOne
.proc AtnOne
    sta atn_buffer
    lda #1
    sta atn_bytes
    lda #0
    sta atn_index
    rts
.endproc
```

«constants»+

```
UNLISTEN = $3F
```

If the byte sent successfully, then we increment our index, and kill the write if needbe.

«ser_write»+

```
GoodWrite:
    ldx ser_index
    inx
    cpx ser_bytes
```

```
    beq WriteDone
    stx ser_index
    jmp XferLoop
```

If we finished the write, then we also need to send the `UNLISTEN` command to indicate final transfer, in addition to clearing out the serial xfer.

«ser_write»+

```
WriteDone:
    lda #0
    sta ser_bytes
    sta ser_index
    lda #UNLISTEN
    jsr AtnOne
    jmp XferLoop
```

Now, let's handle the actual TryWrite routine.

«subrs»+

```
    .export TryWrite
.proc TryWrite
<<try_write>>
WriteDone:
    rts
.endproc
```

First, we signal we're ready to send a byte.

«try_write»=

```
    jsr ClkOff
```

Now, we wait for the readers to become available, keeping in mind if we timeout, we should exit now.

This is handled via a subroutine, that conveniently also returns the timeout condition with carry clear.

«try_write»+

```
    jsr WaitWrite
    bcc WriteDone
```

If this is the end-of-stream, we should delay 256 microseconds here.

«try_write»+

```
    bit ser_eof
    bpl NoEof
    lda ser_index
    cpx ser_bytes
    bne NoEof
```

```
    jsr Wait256Us
NoEof:
```

Now, send each bit!

«try_write»+

```
    ldx #8
WriteLoop:
    jsr ClkOn
    jsr DataOff
    lsr byte_buffer
    bcc WriteZero
    jsr DataOn
WriteZero:
    jsr Wait60Us
    jsr ClkOff
    jsr Wait60Us
    dex
    bne WriteLoop
    jsr ClkOn
    jsr DataOff
    jsr Wait1kUs
```

And, we're done!

Serial Reads *«ser_read»=*

```
    brk ; TODO!
```

Entry and Exit We want to install our IRQ handler, and make sure that whatever IRQ handler that *was* there previous will still be called.

«setup_irq»=

```
    .code
    .export SetupIrq
.proc SetupIrq
    php                     ; save interrupt flag
    sei                     ; disable interrupts
    lda IRQ_VECTOR          ; save the current IRQ handler
    sta old_irq
    lda IRQ_VECTOR+1
    sta old_irq+1
    lda #<MyIrq             ; store ours!
    sta IRQ_VECTOR
    lda #>MyIrq
    sta IRQ_VECTOR
    plp                     ; restore interrupt flags, since they're now safe to occur
```

```
    rts
.endproc
```

«constants»+

```
IRQ_VECTOR = $0314 ; pointer to IRQ service routine
```

«wrapup_irq»=

```
XferDone:
    jmp (old_irq)           ; run the regular IRQ handler now that we're finished!
```

The 6502 has a bug: if we jump indirectly via a variable, and that variable straddles a page – the hibyte in $xxFF and the lobyte in $xy00 – then it'll jump to the wrong address. We'll align the variable just to be safe.

«variables»+

```
    .bss
    .align 2 ; avoid indirect jump bug
old_irq: .res 2
    .export old_irq
```

Low Level Routines Let's start with the easy stuff, the Wait routines. They wait in microseconds, and the CPU clock is conveniently measured in those! Althought, it takes 12 cycles/microseconds to get in and out of a subroutine.

«subrs»+

```
    .export Wait60Us
.proc Wait60Us
    ; 6 cycles to JSR here
    ldy #9      ; +2 cycles=8
Loop:
    dey         ; +2 cyles
    bne Loop    ; +3 cyles while taken, +2 when falling thru
    ; we ran the loop 9 times, the first 8 took 5 cycles, the last took 4.
    ; 8+8*5+4=52
    nop         ; +2 cycles=54
    rts         ; +6 cycles = 60
.endproc
```

«subrs»+

```
    .export Wait256Us
.proc Wait256Us
    ; 6 cycles to JSR here
    ldy #48        ; +2
Loop:
    dey             ; +2
    bne Loop        ; +3 cycles while taken, +2 when falling thru
```

```
    ; last loop thru took 4 cycles, rest took 5
    ; 8+5*(y-1)+4=250
    ; 5*(y-1)=250-8-4
    ; y-1=238/5
    ; y=47.6+1
    ; y=48
    ; 8+5*47+4=247
    nop             ; +2 cycles=249
    rts             ; +6 cycles=255, which is close enough
.endproc
```

«subrs»+

```
    .export Wait1kUs
.proc Wait1kUs
    ; 6 cycles to JSR here
    ; 1000/255=3.9
    jsr Wait256Us   ; +255=261
    jsr Wait256Us   ; +255=516
    jsr Wait256Us   ; +255=771
    ; 229 cycles remaining
    ldy #74        ; +2 cycles=773
Loop:
    dey             ; +2 cycles
    bne Loop        ; +3 cycles every loop thru until last, which takes +2
    ; 773+3*(y-1)+2=1000-6
    ; 3*(y-1)=1000-6-773-2
    ; y-1=219/3
    ; y=73+1
    ; 773+3*(74-1)+2=994
    rts             ; +6 cycles on exit
.endproc
```

The Serial Signal Routines I can never remember the Commodore serial bus signals – ATN, CLK, DATA – should be active 1 or active 0. Hence, my usage of the generic `ClkOn`/`ClkOff` style commands thruout.

Now, it's time to actually write them. I still can't remember as I write this, so I'll define a flag for which way the bits should go. That way, it's easy to change if I mix it up.

«constants»+

```
TRUE = $FF
FALSE = 0
ACTIVE_HI = FALSE
```

I'll also define the register used on the Complex Interface Adapter (CIA for short, *haha*) that I use to access these bits.

«constants»+

```
CIA_PORT = $DD00
```

And the bits themselves.

«constants»+

```
ATN_OUT = 1<<3
CLK_OUT = 1<<4
DATA_OUT = 1<<5
CLK_IN = 1<<6
DATA_IN = 1<<7
```

Now for the macros to set/reset the bits.

«macros»=

```
.if ACTIVE_HI = TRUE
    .mac bit_on bitf
        lda CIA_PORT
        ora #bit
        sta CIA_PORT
    .endmac
    .mac bit_off bit
        lda CIA_PORT
        and #<~bit
        sta CIA_PORT
    .endmac
.else
    .mac bit_on bit
        lda CIA_PORT
        and #<~bit
        sta CIA_PORT
    .endmac
    .mac bit_off bit
        lda CIA_PORT
        ora #bit
        sta CIA_PORT
    .endmac
.endif
```

Now for the actual routines themselves.

«subrs»+

```
.code
AtnOn:
    bit_on ATN_OUT
```

```
    rts
AtnOff:
    bit_off ATN_OUT
    rts
ClkOn:
    bit_on CLK_OUT
    rts
ClkOff:
    bit_off CLK_OUT
    rts
DataOn:
    bit_on DATA_OUT
    rts
DataOff:
    bit_off DATA_OUT
    rts
.export AtnOn,AtnOff,ClkOn,ClkOff,DataOn,DataOff
```

We'll also have a set of routines for handling the input side of the port.

This routine waits for the readers to be ready, returning early if we've taken too much time. As mentioned earlier, this early exit is indicated by returning with carry clear; carry is set if we're good to transfer. Since the `DATA_IN` bit is in bit7 of the port, we can check it via the `BIT` instruction, which will place bit7 into the negative flag.

«constants»+

```
WRITE_ON_DATA_LO = TRUE
```

It turned out that the `.if` didn't work if `WaitWrite` was defined as a *CA65* `proc` (which allow for local labels), so I wrote it as a plain label with an anonymous branch target instead.

«subrs»+

```
.code
.export WaitWrite
WaitWrite:
    .assert DATA_IN = $80,error,"DATA_IN isn't in bit7"
    jsr CheckScanline
    bcc :+
    bit CIA_PORT
    .if WRITE_ON_DATA_LO = TRUE
        bmi WaitWrite
    .else
        bpl WaitWrite
    .endif
    sec ; we're good to write!
```

```
:
    rts
```

Scanline Checks We use the current scanline number to figure out if we've timed out the transfer. Theoretically, we could end up *past* our target scanline, which is a problem when the scanline count wraps around pretty easily.

To handle this, we first check if `target_line` is less than `start_line`. If not, we can perform a normal range check `cur_line >= start_line && cur_line < target_line`. But if they are, we replace the and with an or: `cur_line >= start_line || cur_line < target_line`. If either check succeeds, then we haven't timed out yet.

Let's work through some test cases to check.

- If we start on line 10, and our target line is 102:
  - If we start our check still on line 10, we don't timeout since cur_line >= `start_line`.
  - If we check on lines 11-101, we don't timeout since `cur_line` < `target_line` and >= `start_line`.
  - If we check on lines 102-255, we successfully timeout, since `cur_line` >= `start_line` and not < `target_line`.
  - If we check on lines 0-9, we successfully timeout, since `cur_line` is not >= `start_line`. If we used an `||` instead of an `&&`, we would fail to timeout – since we're still < `target_line`.
- If we start on line 254, and our target line is rolled over to 90:
  - If we start our check still on line 254, we don't timeout since `cur_line` >= `start_line`.
  - If we check on line 255, we don't timeout, since `cur_line` >= `start_line`. If we used an `&&` instead of an `||`, we'd accidentally timeout here – since `cur_line` is not < `target_line`.
  - If we check on lines 0-89, we don't timeout, since `cur_line` < `target_line`. If we used an `&&` instead of an `||`, we'd accidentally timeout here – since `cur_line` is not >= `start_line`.
  - If we check on lines 90-253, we'd successfully timeout, since `cur_line` is not < `target_line` *or* >= `start_line`. "'

«subrs»+

```
; Return carry clear if we've timed out on our xfer time, carry set otherwise.
.export CheckScanline
.proc CheckScanline
    lda target_line
    cmp start_line
    bcc OrCheck
AndCheck:
    lda VIC_SCANLINE
    cmp start_line
```

```
    bcc TimeOut
    cmp target_line
    bcc TimeIn
TimeOut:
    clc
    rts
OrCheck:
    lda VIC_SCANLINE
    cmp start_line
    bcs TimeIn
    cmp target_line
    bcs TimeOut
TimeIn:
    sec
    rts
.endproc
```

The Test Itself

So, let's test something!

We'll use the label `Start` as our *main* entry point.

«main»=

```
    .export Start
.proc Start
<<init-test>>
<<run-test>>
.endproc
```

We'll intialize by clearing RAM, then setting up our IRQ.

«init-test»=

```
<<clear-bss>>
    jsr SetupIrq
```

«clear-bss»=

```
    lda #<__BSS_LOAD__
    sta ser_pointer
    lda #>__BSS_LOAD__
    sta ser_pointer+1
    ldy #0
BssLoop:
    lda ser_pointer+1
    cmp #>(__BSS_LOAD__+__BSS_SIZE__)
    bne BssClear
    lda ser_pointer
```

```
    cmp #<(__BSS_LOAD__+__BSS_SIZE__)
    beq BssDone
BssClear:
    tya                     ; Y=0
    sta (ser_pointer),y
    inc ser_pointer
    bne BssLoop
    inc ser_pointer+1
    jmp BssLoop
BssDone:
    ldx #0
    tya
ZpLoop:
    sta (<__ZEROPAGE_LOAD__),y
    inx
    cpx #<__ZEROPAGE_SIZE__
    bne ZpLoop
    .import __BSS_LOAD__,__BSS_SIZE__,__ZEROPAGE_LOAD__,__ZEROPAGE_SIZE__
```

To actually run the test, we'll start by writing an ATN command to open a disk file.

«data»=

```
    .rodata
open_msg:
    .byte LISTEN+8,OPEN+2,"@0:FOO,S,W"
open_msg_end:
    .export open_msg, open_msg_end
```

«constants»+

```
OPEN = $F0
```

«run-test»=

```
    ldx #0
OpenLoop:
    lda open_msg,x
    sta atn_buffer,x
    inx
    cpx #open_msg_end-open_msg
    bne OpenLoop
    lda #0
    sta atn_index               ; clear byte index into ATN array
    stx atn_bytes               ; write the length of the message
```

We'll wait for the ATN command to be finished.

«run-test»+

```
    jsr WaitATN
```

«subrs»+

```
    .export WaitATN
.proc WaitATN
    lda atn_bytes
    bne WaitATN
    rts
.endproc
```

Now, we'll write the file data.

«data»+

```
file_data:
    .byte "BAR",0
file_data_end:
    .export file_data,file_data_end
```

«run-test»+

Building

Let's put together our whole source file.

«atntest.s»=

```
<<constants>>

<<macros>>

<<variables>>

<<irq>>

<<setup_irq>>

<<subrs>>

<<main>>

<<data>>
```

CA65 requires a linker config file.

«atntest.cfg»=

```
MEMORY {
    PRG_ADDR: start = $0000, size = 2;
    RAM: start = $801, size = $97FF;
    ZEROPAGE: start = $FB, size = 4, file = "";
}

SEGMENTS {
    ZEROPAGE: load = ZEROPAGE, type = zp, define = yes;
    PRG_ADDR: load = PRG_ADDR, type = ro;
    BASIC_HEADER: load = RAM, type = ro;
    CODE: load = RAM, type = ro;
    RODATA: load = RAM, type = ro;
    DATA: load = RAM, type = rw;
    BSS: load = RAM, type = bss, define = yes, align = 2;
}
```

We need to insert the boot header.

«boot.s»=

```
    .segment "PRG_ADDR"
    .word $801
    .segment "BASIC_HEADER"
    .word null_line
    .word 10
    .byte $9E,"2061",0
null_line:
    .word 0
    .assert *=2061,error,"bad BASIC header"
    .import Start
    jmp Start
```

Let's make our Makefile:

«makefile»=

```
OBJECTS = boot.o atntest.o

atntest.prg atntest.ll: $(OBJECTS) atntest.cfg
    ld65 -o atntest.prg -C atntest.cfg $(OBJECTS)

%.o: %.s
    ca65 -o $@ $*.s
```

But, just for fun, we'll add a make target to generate a nice PDF via pandoc.

«Makefile»=

```
.PHONY: all clean pdf
```

```
all: atntest.pdf atntest.prg

clean:
    rm -f atntest.prg atntest.ll $(OBJECTS) atntest.pdf

pdf: atntest.pdf

atntest.pdf: atntest.md
    pandoc -o atntest.pdf --filter pandoc-annotate-codeblocks atntest.md
```

<<makefile>>

Conclusion