« How many Commodore 64 computers were really sold?
Leave security to security code. Or: Stop fixing bugs to make your software secure! »

A 256 Byte Autostart Fast Loader for the Commodore 64

Platforms like the Commodore 64 are still a lot of fun to work with, not only because the limitations make certain tasks a real challenge, but also because it is possible to use many interesting tricks on a bit- and cycle-level – after all, the system is well-understood and practically all setups were identical.

This article presents a C64 "fast bootloader": A small program that auto-starts when loaded into memory and chain-loads e.g. a game, but replacing the slow disk transfer routines in ROM with much faster ones – and all this fits into a single 256 bytes sector.

The C64 and the Drive 1541

The C64 is an 8-bit computer released in 1982 that is powered by a 1 MHz 6502-based CPU and has 64 KB of RAM. A typical C64 setup also consists of a Commodore 1541 disk drive (5.25″ SS/DD media, 170 KB per side) which is a 1 MHz 6502 computer with 2 KB of RAM itself.

The IEC Bus

The computer and the drive are connected through a serial cable that carries three lines from the computer to the drive (ATN, CLK, DATA) and two from the drive to the computer (CLK, DATA). This IEC bus supports several daisy-chained disk drives (and printers), and the computer uses the ATN line to arbitrate the bus.

IEC was introduced in the C64's predecessor VIC-20 and its floppy drive 1540, as a cheaper version for the parallel IEE-488 bus. Each device had a 6522 "VIA" I/O controller that could do a simple serial protocol in hardware – in theory. The serial port in the VIAs never worked, so Commodore decided to work around the issue by just implementing the protocol in software, after all, both devices had programmable CPUs. The C64, running basically the same system software as the VIC-20, and the 1541, a slightly updated disk drive for the C64, inherited this design.

As a result of every single bit having to go through a software handshake, the ROM code in the C64 and the drive could only transfer about 400 bytes per second. A game that fills all of RAM would take more than two minutes to load. Another issue of the IEC code in the C64 ROM is that it turns off interrupts every once in a while, making it impossible to properly play music in the background while loading data from disk.

Fast Loaders

Practically every game and every demo therefore only had a small boot program that was loaded by the original ROM code, typically less than a kilobyte in size, which contained more efficient serial code for the computer side, as well as corresponding code for the drive, which was uploaded into the drive's RAM using the original serial protocol.

The Protocol

The original IEC protocol uses one clock and one data line in each direction. The sender alternates the clock line on every bit of data, and the receiver has to acknowledge the receipt by alternating its clock line. The idea of a faster protocol is to send a whole byte, bit by bit without a clock signal: Whenever both devices are undisturbed by interrupts and device DMA, we can assume the clocks of both devices to be enough in sync for the duration of transmitting a byte. (In fact, the 1541 is clocked a little bit faster than a PAL C64, so there is one extra 1541 cycle for every 67 C64 cycles.) Since the clock line is now not necessary for the handshake, it can be used for data, so we can transmit two bits at a time, and transfer a byte in four steps.

Receiving a Byte

The IEC bus is controlled through port A of the C64's second 6526 "CIA" I/O chip, which is accessible through the MMIO address $DD00:

7	IEC DATA IN
6	IEC CLK IN
5	IEC DATA OUT
4	IEC CLK OUT
3	IEC ATN Signal OUT
2	RS-232
1-0	VIC Select

We can signal the drive when we are ready to receive a byte through bits 4 and 5, and the drive can send data through bits 7 and 6. We need to be careful not to change bits 0 and 1, since these select the 16 KB memory bank that the video chip fetches its data from – typically, these bits are both 1.

The fastest way to receive a byte from the serial bus (given the sender is fast enough) is to repeatedly read two bits from $DD00, shift them down, then read the next two bits, and repeat all this four times:

```
lda $DD00 ; get 2 bits into bits 6-7
```

```
    lsr
    lsr       ; move down into bits 4-5
    eor $DD00 ; get 2 more bits
    lsr
    lsr       ; move everything down to bits 2-5
    eor $DD00 ; get 2 more bits
    lsr
    lsr       ; move everything down to bits 0-5
    eor $DD00 ; get last 2 bits
```

The trick here is to XOR the new bits onto the already received and shifted bits, this way we avoid shifting, ANDing and ORing. An absolute load taking four cycles and a shift instruction two, receiving a byte takes 28 cycles total. We need to make sure the sending side has the same timing.

Sending a Byte

The 1541 disk drive has the IEC bus exposed through port A of its first "VIA" I/O chip, which is mapped at $1800:

7	IEC ATN IN
6-5	Device number jumper
4	IEC ATN ACK OUT
3	IEC CLOCK OUT
2	IEC CLOCK IN
1	IEC DATA OUT
0	IEC DATA IN

The bits to write the data into are number 1 and number 3, which are not next to each other, so sending is a little more complicated than receiving. The following code assumes that the low 4 bits of the data byte are in register A, and the high 4 bits are in register Y:

```
    sta $1800 ; send bits 1 and 3 of A
    asl       ; bits 0 and 2 become bits 1 and 3
    and #$0F  ; mask off bit #4
    sta $1800 ; send bits 0 and 1 in A
    tya
    nop
    sta $1800 ; send bits 1 and 3 in Y
    asl       ; bits 0 and 2 become bits 1 and 3
    and #$0F  ; mask off bit #4
    sta $1800 ; send bits 0 and 1 in Y
```

The idea is to first just write the low 4 bits into the output port, therefore sending bits 1 and 3. Shifting the value left by one will put bits 0 and 2 into positions 1 and 3 and we can send them by writing them into the output port. Then we repeat this with the upper four bits.

The absolute stores are taking 4 cycles each, and "asl", "and", "tya" and "nop" all take 2 cycles each, so this code has exactly the same timing as the receiver side.

But unfortunately, this code sends the bits in the wrong order, so we need to correct it, either on the sending or on the receiving side. A simple lookup in a 256 byte table (on either side) would do the job, but storing the table in the fast loader would increase its size significantly – and therefore the time to load the fast loader with the original Commodore serial code. It would be possible to generate this table at runtime, but a good tradeoff between size and performance is this code (34 bytes; with a few bytes more, it could be converted into a generator for the table):

```
    eor #3        ; fix up bits 0-1 (VIC bank)
    pha           ; save original
    lsr
    lsr
    lsr
    lsr           ; get high nybble
    tax           ; to X
    ldy enc_tab,x ; super-encoded high nybble in Y
    pla
    and #$0F      ; lower nybble
    tax
    lda enc_tab,x ; super-encoded low nybble in A

enc_tab:
    .byte %1111, %0111, %1101, %0101, %1011, %0011, %1001, %0001
    .byte %1110, %0110, %1100, %0100, %1010, %0010, %1000, %0000
```

First, we invert the lowest two bits of the value to send, because the receiving side always reads back "11" in the lowest bits of $DD00. Then we encode both the high and the low 4 bits using a 16 byte table, putting the results in the A (low) and Y (high) registers. This table interleaves the four bits so that 0123 becomes 3120 and inverts every bit: An interesting property of the lines from the drive to the computer is that all lines arrive inverted at the computer side. This is not the case for the lines from the computer to the drive. So after this conversion, the send code above can blindly send the bits, which will show up as the original value on the receiver side.

Handshake

But it is not enough to just bang the bits on the bus – after every byte, we need to do a handshake. In a perfect world, we could just send a complete sector (256 bytes on a 1541) in a go, maybe even in an unrolled loop for extra speed, but there are several reasons against it. One reason is that the floppy is clocked about 2% faster, so we're off by one cycle every 67 cycles. The fastest possible send loop is about 50 cycles per byte (including the byte encoding), and the time between two pieces of data on the bus (i.e. $1800 writes/$DD00 reads) is 8 cycles, so after 8*67 = 536 cycles = 10 transfered bytes we have missed two bits. Getting the timing correct across such a long time becomes very tricky then.

Another problem is the fact that the video chip in the C64 ("VIC-II") requires 40 cycles for DMA on every 8th line of the visible screen that it sends to the display, completely stopping the CPU, which would mess up all timing. One raster line on the C64 is exactly 63 cycles long, so these CPU stalls ("badlines") happen every 504 cycles. If we start a transfer of several bytes just after one of these badlines, we have time for a maximum of about 9 bytes ((504-40)/50). Or we could only transfer data while the VIC is outside the visible area, but this is only in 112 of the 312 lines, so we would be wasting about 64% of the processing power. Most fast loaders just turn off the screen, so the VIC doesn't do these fetches any more, but we don't want to take this shortcut!

Badlines happen every time the vertical raster location (readable in register $D012 of the VIC) is between 50 and 249 and the the lowest 3 bits reach the value of 3. (Actually, this value is variable and corresponds to the lowest 3 bits in register $D011.) So the first thing to check is whether we are below 50 or above 249 (i.e. between 250 and 311). $D012 only holds the lowest 8 bits of the raster register (the MSB is stored in the MSB of $D011), but just checking for $D012 being below 50 already means that the raster register is between 0-49 or 256-305 – this easier check with some false positives is preferrable to a more compicated but slower check. So if we are in the visible area, we must watch out whenever we are are in a line that ends in "2", because a badline will happen some time within the next 63 cycles. In every other case, a badline won't happen within at least 63 cycles, so we are safe to spend our 28 cycles in the receiver code. The following code does this:

```
wait_raster:
    lda $D012           ; vertical raster position (bits 0-7)
    cmp #50             ; between 0-49 or 256-305?
    bcc wait_raster_end ; yes, so it's safe
    and #$07            ; lowest 3 bits
    cmp #$02            ; are we in the line before a badline?
    beq wait_raster     ; yes, then wait until we are not
wait_raster_end:
```

If sprites are visible on the screen, this also requires extra DMAs from the VIC, but we assume that there no sprites active. If in doubt, writing 0 into VIC register $D015 makes sure they are all turned off.

Now the actual handshake is rather easy. The protocol is this: At the beginning, both the computer and the drive set their handshake flags to "not ready". When the drive has data in its buffer and is ready to send a byte, it sends its flag to "ready". Then the C64 makes sure it is not in danger of a badline and sets its "ready" flag. Just after the transfer of the byte, both devices set their flags to "not ready" again. The drive signals readyness with CLK=0, and the computer does so with both CLK=0 and DATA=0, so the code looks like this:

```
;-----
; C64 at initialization time
    lda #VIC_OUT | DATA_OUT ; CLK=0 DATA=1
    sta $DD00               ; we're not ready to receive

;-----
; drive at initialization time
    lda #F_CLK_OUT          ; CLK=1 DATA=0
    sta $1800               ; drive code running, we're not ready to send

;-----
; C64 waiting for drive code running
wait_fast:
    bit $DD00
    bvs wait_fast           ; wait for CLK=1, i.e. drive code running

;-----
; drive when it is ready to send (i.e. byte in buffer and converted)
```

```
    lda #0                  ; CLK=0 DATA=0
    sta $1800               ; we're ready to send

;-----
; C64 waiting for drive ready to send
wait_byte:
    bit $DD00
    bvc wait_byte           ; wait for CLK=0, i.e. drive ready to send

;-----
; C64 when it is ready to receive (i.e. not in danger of badline)
    lda #VIC_OUT            ; CLK=0 DATA=0
    sta $DD00               ; we're ready, start sending!

;-----
; drive waiting for C64 ready to receive
wait_c64:
    ldx $1800
    bne wait_c64            ; needs all 0

;-----
; C64 after receiving a byte
    lda #VIC_OUT | DATA_OUT ; CLK=0 DATA=1
    sta $DD00               ; not ready any more, don't start sending

;-----
; drive after sending a byte
    jsr $E9AE               ; CLK=1 (use ROM code to opimize for size)
```

Please note that logic on all $DD00 reads on the computer side looks backwards, because the bits get inverted.

Timing after the Handshake

The send and the receive code have exactly the same timing, but we need to make sure that they also start at the same time. The C64 code cannot start reading data from the bus directly after telling the drive that it is ready to receive, because the drive is testing for the C64's readiness in this loop:

```
;-----
; drive waiting for C64 ready to receive
wait_c64:
    ldx $1800
    bne wait_c64            ; needs all 0
```

The load takes 4 cycles, and the branch 2 or 3, depending on whether it is taken. The actual bus access for the read from $1800 takes place in the third cycle, so in the worst case, the computer signals that its ready exactly the fourth cycle: In this case, the LDX has read the old value and the branch is taken, the LDX reads the value again, gets the right value now, and doesn't take the branch. So the maximum time until the 1541 reacts is 10 cycles: 1 for the last cycle in the LDX, 3 for the taken branch, 4 for another LDX, and 2 for the final non-taken branch. This is the code that delays for 10 cycles between the ready signalling and the reading of the first 2 bits:

```
    lda #VIC_OUT ; CLK=0 DATA=0
    sta $DD00    ; we're ready, start sending!
    pha          ; 3 cycles
```

```
pla         ; 4 cycles
bit $00     ; 3 cycles
lda $DD00   ; get 2 bits into bits 6&7
```

Reading Sectors

Now that we have the transfer code for one byte in place, we can easily construct a loop on both sides that repeats the transfer 256 times for a full sector. But we also need code running inside the drive that reads sectors from disk in the first place. The easiest way to do this is use the ROM code. It will happily position the read head for us, wait for the sector to come by, read it, decode the on-disk bit encoding (6-to-4 Group-Code-Recording, GCR) and put it into a buffer:

```
lda #TRACK
sta $06
lda #SECTOR
sta $07
lda #0
sta $f9     ; buffer at $0300
cli
jsr $D586   ; read sector
sei
```

The 1541 has 5 buffers, numbered 0 through 4, from $0300-$03FF to $0600-$06FF. The track and sector for buffer 0 are stored in zero page addresses 6 and 7, the ones for buffer one in addresses 8 and 9 and so on. Note that during the whole process of loading data into the C64, we have interrupts disabled ("SEI"), so we need to reenable them while reading from disk for the timers to work properly.

More advanced fastloaders don't use the ROM code for reading, but implement a more optimized version, achieving another minor speedup. Bigger speedups can be achieved by changing the algorithm of reading completely: Tracks on a 1541 disk are up to 21 sectors, 256 bytes each, but the 1541 RAM is only 2 KB, so it cannot read a whole track into memory. Therefore it reads one sector, transfers it, reads another one and so on. After reading a sector, it needs to be decoded and sent, during which time the disk continues spinning, causing the drive to miss a few sectors. So it would be a bad idea to store files on consecutive sectors, since this would mean it has to wait for a whole turn of the disk (one fifth of a second) for that sector to arrive under the head again.

Instead files are stored in an interleaved fashion, typically with an interleave factor of 4, meaning a file is for example stored on sectors 0, 4, 8, 12 etc. When a fast loader is used, it would typically require a different interleave factor for optimal performance, but unfortunately the interleave factor is a property of the already written disk. A very advanced method of fast loading is therefore to always read the sector that comes by next and transfer it, unless it has been transfered before, until the complete track is in the C64. The C64 then sorts the sectors in the correct order. For smaller files, this does not work too well, since this method always reads and transfers complete tracks.

Even different fast loaders (like Heureka Sprint, used by Turrican and some other Rainbow Arts titles) require the data on disk to be encoded differently, making decoding more efficient. Some copy programs, like “Master Copy” don’t decode the sector data at all – but they can only do this because they write the same encoding, and the actual payload data is never required.

But in order to keep the implementation really small (custom read code is in the order of 600 bytes) let’s stay with the code in ROM.

Uploading the Code into the Drive

Let’s consider both pieces of code on the C64 and the 1541 side finished now, but what’s still missing is code to upload the drive code into the RAM of the 1541 and run it. There are several ways of doing this: The 1541 operating system over the original IEC protocol has a “memory-write” (“M-W”) command, allowing us to upload up to 36 bytes at a time, and a “memory-execute” (“M-E”) command that makes the CPU jump to the address we specify. Our 1541 code is about 100 bytes, which would take about a quarter of a second to upload with the original protocol.

But there is a way to avoid this cost: All code and data in the C64 came originally from disk, so why would we download it to the C64 and upload it to the 1541 again? We can just instruct the 1541 to read a sector and execute it. This can be done with block-read (“B-R”) and “memory-execute”, or with the specialized instruction “block-execute” (“B-E”). Unfortunately, “block-execute” does not work on the concept of buffers, but on the concept of channels which abstract buffers, making this more complex than it would have to be.

A common trick is to upload minimal 6502 code to the drive that reads a sector and jumps to it and execute that. And it’s even possible to avoid the “memory-write” command: When sending the “memory-execute”, we can send trailing bytes, for a command that is up to 42 bytes long. The code would just travel with the “memory-execute” command, and the execution address would point to this very code in the temporary command buffer:

```
    lda #$0f
    sta $b9   ; secondary address
    sta $b8   ; logical file number
    ldx #cmd
    lda #cmd_end - cmd
    jsr $fdf9 ; filnam
    jsr $f34a ; open
    brk

cmd:
    .byte "M-E"
    .word $0200 + cmd_code - cmd
cmd_code:
    lda #18   ; track 18, sector 18
    sta $08
    sta $09
    lda #1    ; buffer at $0400
    sta $f9
    jsr $d586 ; read sector
    jmp $0400 ; jump to the code we loaded
cmd_end:
```

The command buffer in the 1541 is located at $0200, so the "memory-execute" jumps to the first byte just after the command itself, at $0205. We choose to store the floppy code on track 18, sector 18: Track 18 is decidated to directory entries, so unless the disk has 144 files on it, it is unlikely all sectors of track 18 are in use. Reading the code from track 18 also means the head does not have to move if we store the C64 loader there, too.

Fitting C64 and Drive Code Into a Single Sector

But there is an even simpler and faster solution: If we manage to fit both the C64 code and the floppy code into a single sector, we don't have to read another sector, but we can just send a "memory-execute" into the buffer that the block was loaded into:

```
cmd:
    .byte "M-E"
    .word $0482
cmd_end:
```

The default buffer is #1 at $0400 in the drive's memory, so after the start program got loaded into the C64, the sector can still be found at $0400.

The default 1541 file system does not support random file access, therefore there is no central data structure that allows a lookup of the sector number following the current one. Instead, the link to the next track and sector is stored in the first two bytes of every sector, reducing the usable space in a sector to 254 bytes (and making seeks in a file very expensive). So the first byte in a sector is the track (1-35) and the second byte is the sector (0-20) of the following block. If it is the last block of a file, the track number is zero and the sector field contains the number of valid bytes in the block; all bytes afterwards will be ignored and not transfered. This allows files that are not a multiple of 254 bytes in size.

So the trick is to create a file that is about half a sector in size and contains the C64 code, and we store the drive code in the unused half of the sector. So the reason why we always optimized for code size when choosing algorithms before was because we really need to fit everything in 256 bytes!

Header

Now what is the executable file format, you may ask? What are the headers, how are they structured? How much data is used for headers? It is complicated.

The shell of the C64 was Commodore BASIC, a derivative of Microsoft BASIC for 6502. So you would load BASIC programs from disk with the "LOAD" command, you could have them printed on the screen with "LIST" and edit them; and if you wanted to run them, you would type "RUN". This concept wasn't really meant for programs not written in BASIC, but it was enough to have a small BASIC header in front of your assembly program, like this:

```
10 SYS2061
```

BASIC programs get loaded to $0801, so the machine code is stored directly after this small BASIC header which tells the interpreter to run machine code at 2061 = $080D. But this wastes 12 bytes and requires the user to type "RUN" after the program is loaded.

Autostart

It is much nicer to have the program autostart directly after the "LOAD" command. The trick here is to have the program load not into BASIC RAM, but into a region where it overwrites vectors – it's basically a buffer exploit! Here is a rough memory map of the C64:

$0000-$00FF	BASIC and KERNAL variables
$0100-$01FF	Stack
$0200-$0258	BASIC input buffer
$0259-$02FF	BASIC and KERNAL variables
$0300-$033B	System vectors
$033C-$03FF	I/O buffer
$0400-$07FF	Screen RAM
$0800-$9FFF	BASIC RAM
$A000-$BFFF	BASIC ROM
$C000-$CFFF	RAM
$D000-$DFFF	Device MMIO
$E000-$FFFF	KERNAL ROM

Commonly, autostart programs would overwrite the system vectors at $0300:

$0314-$0315	IRQ vector
$0316-$0317	BRK vector
$0318-$0319	NMI vector
$031A-$031B	OPEN vector
$031C-$031D	CLOSE vector
$031E-$031F	CHKIN vector
$0320-$0321	CHKOUT vector
$0322-$0323	CLRCHN vector
$0324-$0325	CHRIN vector
$0326-$0327	CHROUT vector
$0328-$0329	STOP vector
$032A-$032B	GETIN vector
$032C-$032D	CLALL vector
$032E-$032F	unused
$0330-$0331	LOAD vector
$0332-$0333	SAVE vector
0334-033B	unused
033C-03FB	Tape buffer

Your program would load to $0326, for example, overwriting the CHROUT vector as well as the 5 following vectors, and your code would be loaded into the tape buffer starting at $033C. When loading is finished, the BASIC interpreter wants to print "READY.", jumping over the CHROUT vector at $0326 and therefore into your code.

The problem with this solution is that we have to preserve the values of some of the vectors between $0328 and $033B, because the original LOAD code in ROM calls the STOP vector to test whether the user pressed the STOP key. So our file would have to contain the original values, not only wasting 12 bytes, but also introducing potential incompatibilities if the user has a cartridge like the Final Cartridge III or the Action Replay VI attached – these devices were practically ROM extensions and hooked some of these vectors to provide improved functionality.

(Actually, overwriting the STOP vector is useful in a different scenario: This way, we can catch execution during the load operation as opposed to after it and continue loading the same file with a replacement bus protocol.)

A different way to gain control after loading is to load into the stack and overwriting the address returned to after the LOAD is finished. The stack on the 6502 is always located between $0100 and $01FF, so if we overwrite this complete area with a value of 2, we would put all "$0202" vectors on the stack, catching execution as soon as the inner ROM LOAD code returns to its caller. Since the 6502 increments the return address after it fetches it from the stack, our payload would live at $0203, which is still pretty much directly after the stack area. But of course overwriting the complete stack is a waste: Experimentation shows that the one vector on the stack that actually counts is located at $01F8/$01F9.

Laying Out the Code

The problem with the payload starting at $0203 is that we can only use the memory up to $0258 (55 bytes) – this is the buffer for a BASIC input line. Unfortunately, this is not enough, since our code is more like 110 bytes. We can put the payload before the vector we overwrite, i.e. onto the stack. But we must be careful, because the LOAD code in ROM uses some stack, overwriting the area between $01ED to $01F7. So let's have our code start somewhere in the stack area, going up to $01EC, and put a JMP to the code at $0203 to catch the stack return.

The 11 bytes at $01ED-$01F7 (stack that gets overwritten while loading) and the 9 bytes at $01FA-$0202 (area between the vector on the stack we overwrite and our first instruction at $0203) seems wasted – but not quite. We can use $01FE-$0202 to store our 5 byte "M-E" string, and we just fill all bytes from $01ED to $01FD with "2". This gives us extra safety that our code will work machines with replacement ROMs or extended ROM routines that use a slightly different stack layout – as long as they don't use more stack and overwrite our code.

Final Words

Fast loaders and autostart bootloaders have been around for almost as long as the C64. Fast loaders have used the stack trick before, and 26 cycle drive transfer code with the screen turned on has been in use before as well. So what's really novel about the bootloader described in this article is the combination of the most optimized tricks into a single-block (256 byte) program. That's the beauty of programming for the C64: Practically everything is implicitly open source, since the best algorithms fit in a few hundred bytes of code, and an experienced C64 hacker can reverse-engineer existing code and incorporate it into his own. That's how it has always been done.

The Code

Here is the complete code, which can be assembled with the ca65 assembler of the cc65 compiler suite.

```
TARGET := $0400
TRACK := 18

DATA_OUT := $20 ; bit 5
CLK_OUT  := $10 ; bit 4
VIC_OUT  := $03 ; bits need to be on to keep VIC happy

seccnt = 2

;-----------------------------------------------------------------------
-------
; Hack to generate .PRG file with load address as first word
;-----------------------------------------------------------------------
-------
.segment "LOADADDR"
.addr *

;-----------------------------------------------------------------------
-------
; Send an "M-E" to the 1541 that jumps to floppy code.
; Then receive one block and run it.
; This code lives around $0190.
;-----------------------------------------------------------------------
-------
.segment "PART2"
main:
    lda #$0f
    sta $b9
    sta $b8
    ldx #<memory_execute
    ldy #>memory_execute
    lda #memory_execute_end - memory_execute
    jsr $fdf9 ; filnam
    jsr $f34a ; open

    sei
    lda #VIC_OUT | DATA_OUT ; CLK=0 DATA=1
    sta $DD00 ; we're not ready to receive

; wait until floppy code is active
wait_fast:
    bit $DD00
    bvs wait_fast ; wait for CLK=1 (inverted read!)

    lda #sector_table_end - sector_table ; number of sectors
    sta seccnt
    ldy #0
get_rest_loop:
    bit $DD00
    bvc get_rest_loop ; wait for CLK=0 (inverted read!)

; wait for raster
wait_raster:
    lda $D012
    cmp #50
    bcc wait_raster_end
    and #$07
    cmp #$02
    beq wait_raster
wait_raster_end:

    lda #VIC_OUT ; CLK=0 DATA=0
    sta $DD00 ; we're ready, start sending!
    pha ; 3 cycles
    pla ; 4 cycles
    bit $00 ; 3 cycles
    lda $DD00 ; get 2 bits into bits 6&7
    lsr
    lsr ; move down by 2 (bits 4&5)
    eor $DD00 ; get 2 more bits
    lsr
    lsr ; move everything down (bits 2-5)
    eor $DD00; get 2 more bits
    lsr
    lsr ; move everything down (bits 0-5)
    eor $DD00 ; get last 2 bits, now 0-7 are populated

    ldx #VIC_OUT | DATA_OUT ; CLK=0 DATA=1
    stx $DD00 ; not ready any more, don't start sending

selfmod1:
    sta TARGET,y
    iny
    bne get_rest_loop

    inc selfmod1+2
    dec seccnt
    bne get_rest_loop

inf:
    jmp inf

.segment "VECTOR"
; these bytes will be overwritten by the KERNAL stack while loading
; let's set them all to "2" so we have a chance that this will work
; on a modified KERNAL
    .byte 2,2,2,2,2,2,2,2,2,2,2
; This is the vector to the start of the code; RTS will jump to $0203
    .byte 2,2
; These bytes are on top of the return value on the stack. We could use
; them for data; or, fill them with "2" so different versions of KERNAL
; might work
    .byte 2,2,2,2

.segment "CMD"
memory_execute:
     .byte "M-E"
     .word $0480 + 2
memory_execute_end:
```

```
;---------------------------------------------------------------------------
-------
; Jump to code that receives data.
;---------------------------------------------------------------------------
-------
.segment "START"
    jmp main

;---------------------------------------------------------------------------
-------
;---------------------------------------------------------------------------
-------
; C64 -> Floppy: direct
; Floppy -> C64: inverted
;---------------------------------------------------------------------------
-------
;---------------------------------------------------------------------------
-------

.segment "FCODE"

F_DATA_OUT := $02
F_CLK_OUT  := $08

sec_index := $05

start1541:
    lda #F_CLK_OUT
    sta $1800 ; fast code is running!

    lda #0 ; sector
    sta sec_index
    sta $f9 ; buffer $0300 for the read
    lda #TRACK
    sta $06
read_loop:
    ldx sec_index
    lda sector_table,x
    inc sec_index
    bmi end
    sta $07
    cli
    jsr $D586       ; read sector
    sei

send_loop:
; we can use $f9 as the byte counter, since we'll return it to 0
; so it holds the correct buffer number "0" when we read the next sector
    ldx $f9
    lda $0300,x

; first encode
    eor #3 ; fix up for receiver side (VIC bank!)
    pha ; save original
    lsr
    lsr
    lsr
    lsr ; get high nybble
    tax ; to X
    ldy enc tab,x ; super-encoded high nybble in Y
    ldx #0
    stx $1800 ; DATA=0, CLK=0 -> we're ready to send!
    pla
    and #$0F ; lower nybble
    tax
    lda enc_tab,x ; super-encoded low nybble in A
; then wait for C64 to be ready
wait_c64:
    ldx $1800
    bne wait_c64; needs all 0

; then send
    sta $1800
    asl
    and #$0F
    sta $1800
    tya
    nop
    sta $1800
    asl
    and #$0F
    sta $1800

    jsr $E9AE ; CLK=1 10 cycles later

    inc $f9
    bne send_loop
    beq read_loop

end:
    jmp *

enc_tab:
    .byte %1111, %0111, %1101, %0101, %1011, %0011, %1001, %0001
    .byte %1110, %0110, %1100, %0100, %1010, %0010, %1000, %0000

sector_table:
    .byte 0,1,2,3,$FF
sector_table_end:
```

This is the linker script:

```
MEMORY {
    # hack to get the load address as the first 2 bytes into the .PRG
    LOADADDR: start = $0188, size = 2;

    # the receive code, filled with $02s that overwrite the top few
bytes of
    # the stack and make the KERNAL loader return to $0203
    PART2:    start = $0188, size = $0065, fill = yes, fillval = $FF,
file = %O;

    VECTOR:   start = $01ED, size = $0011, fill = yes, fillval = $FF,
file = %O;

    CMD:      start = $01FE, size = $0005, fill = yes, fillval = $FF,
file = %O;

    # entry point $0203 due to stack overwritten with $02s
    # code that transfers M-E
    START:    start = $0203, size = $0003, fill = yes, fillval = $ff,
file = %O;

    FCODE:    start = $482, size = $007E, fill = yes, fillval = $ff,
file = %O;
}

SEGMENTS {
    LOADADDR:   load = LOADADDR,    type = ro;
    START:      load = START,       type = ro;
    PART2:      load = PART2,       type = ro;
    CMD:        load = CMD,         type = ro;
    VECTOR:     load = VECTOR,      type = ro;
    FCODE:      load = FCODE,       type = ro;
}
```

This script for the c1541 tool, which puts the code into a disk image:

```
format autostart,01
write "start.prg"
```

And this is the shell script that builds the whole thing:

```
ca65 start.s &&
ld65 -C start.cfg start.o -o start.prg &&
dd if=/dev/zero of=autostart.d64 bs=256 count=683 &&
c1541 autostart.d64 < c1541script.txt
```

Note that the c1541 tool creates a file with the whole block on disk, so in practice, the 1541 code will be loaded into the C64 as well, but never used. So the two link bytes of the block would have to be manually changed to decrease its size to achieve maximum speed.

This entry was posted on Monday, February 7th, 2011 at 23:58 and is filed under default. You can follow any responses to this entry through the RSS 2.0 feed. You can leave a response, or trackback from your own site.

16 Responses to “A 256 Byte Autostart Fast Loader for the Commodore 64”

DeeKay says:
8. February 2011 at 2:52

...but.. but... Is it 256 or 254 bytes now? Cause if it's 256, it won't fit into a single sector! ;-)

j.t.d. says:
8. February 2011 at 3:25

I think, 1 sector = 256 bytes, numbered 0-255.

Michael Steil says:

8. February 2011 at 4:08

@DeeKay: 256. :-) I have 254 bytes of code, but the two link bytes are part of my trick to avoid uploading the drive code, so they count towards the total size.

John L says:
8. February 2011 at 6:37

You mentioned earlier that common boot loaders could handle around 300 bytes per second – this is a little more efficient/faster, right ?

Peter Ferrie says:
8. February 2011 at 9:06

Perhaps you can save one byte this way?
Replace
pha ; 3 cycles
pla ; 4 cycles
bit $00 ; 3 cycles
…
ldx #VIC_OUT | DATA_OUT ; CLK=0 DATA=1

with
ldx #VIC_OUT | DATA_OUT ; CLK=0 DATA=1, 2 cycles
lsr $ef, x ;(or somewhere else known to be safe), 6 cycles
txa ;(because A will be destroyed on next instruction anyway), 2 cycles

Nate says:
8. February 2011 at 9:56

John L,

Michael says his routine is about 50 clocks per byte, which would be 50 microseconds/byte or 20,000 bytes/sec on the 1 Mhz 6502.

That is fast, but not as fast as the media rate (40 KB/s). Given interleave and use of the DOS encoding of sectors, the media transfer rate will be the dominating factor now.

V-MAX, for example, used an alternative sector encoding scheme with minimal syncs (10 bits) and cycle-exact processing of data from the media. This saved on GCR conversion time.

Michael's routine is tiny and a great combo of drive/host code. It is not claimed to be the fastest loader ever.

Ingo says:
8. February 2011 at 10:33

Hey, that bit pair timing looks familiar! The 1581 fastloader of the Action Replay uses a shorter time for the first pair, but after that the differences are the same:

static const generic_2bit_t ar6_1581_send_def = {
.pairtimes = {50, 130, 210, 290}, // microseconds*10
.clockbits = {0, 2, 4, 6},
.databits = {1, 3, 5, 7},

.eorvalue = 0
};

Yours would be (untested!):
generic_2bit_t pagetable_send_def = {
.pairtimes = {70, 150, 230, 310},
.clockbits = {0, 2, 4, 6},
.databits = {1, 3, 5, 7},
.eorvalue = 3
};

but says:
9. February 2011 at 0:28

Thank you for this very interesting article!

I tried to assemble start.s but it throws an error at line 26.
Can you fix that?

Michael Steil says:
9. February 2011 at 3:44

@but: Thanks for the feedback. HTML stole a less-than and a greater sign. Fixed, please try again!

Krusty Vader says:
9. February 2011 at 15:31

This kind of articles are killing me… i feel so dumb.

One of this day I'm gonna dust off my 64, my 2 1541 and that old assembler manual.

Damiano says:
11. February 2011 at 15:39

Hi Michael,

may you explain better why you invert the lower 2 bits "First, we invert the lowest two bits of the value to send, because the receiving side always reads back ?11? in the lowest bits of $DD00." Thank you! Damiano

Per Olofsson says:
13. February 2011 at 10:56

@Damiano:

IEC register on the 1541 side:

http://unusedino.de/ec64/technical/aay/c1541/via10.htm

IEC register on the C64 side:

http://unusedino.de/ec64/technical/aay/c64/cia20.htm

1541 IEC schematic:

http://www.zimmers.net/anonftp/pub/cbm/schematics/drives/new/1541/service/Page_09.html

The signals in the 1541 go through a 7406 inverter, which means that if you store a 1 in the 1541, the C64 will read a 0.

MiaM says:
12. March 2011 at 5:08

Interesting, thanks for this!

If the data that gets fastloaded is only going to be read by the fastloader, it could just as well be stored bitswapped directly on disk. This would possibly be a slight speedup itself, and perhaps the freed up code space could be used for even more efficient transmission code (for example sending more than one byte per handshake)?

Joe Cincotta says:
9. April 2011 at 14:01

One of the best commodore articles I have ever read. Thankyou so much – awesome you keep the '64 alive.

Thomas Tempelmann says:
19. June 2011 at 8:27

Impressive.

Back when I reverse engineered the 1541's ROM, none of the details about timing constraints etc. were available. I had to do a lot of guessing and went by trial-and-error. I remember having spent a lot of time trying to speed up the IEC comms, but my code was still quite poor compared to what others made out of it soon after.

Yet, those early times were very exciting, being able to make quite a difference with just a home computer.

BTW, I also documented my Fcopy story here, in case someone is curious: http://stackoverflow.com/questions/193016/reverse-engineering-war-stories

The story of 15 Second Copy for the C-64 « pagetable.com says:
18. July 2011 at 6:18

[…] are floating around on the net. For better understanding on the C64/1541 handshake issues, refer to this article. If you're wondering about the weird bvc * loops: the 6502 CPU of the 1541 has an SO pin, […]

Leave a Reply

Name (required)

Mail (will not be published) (required)

Website

Anti-spam word: (Required)*

To prove you're a person (not a spam script), type the security word shown in the picture.

Click on the picture to hear an audio file of the word.

newlywed

Submit Comment