Creating Self-Booting/Auto-Starting Programs by TFG/Manic Mailman Designs / Oliver Jones

Feel free to correct, improve, extend or delete the stuff below if you know better:

Normally, you'd load and execute a program from disk like that:

LOAD“PRG”,8,1 (+CR) RUN (+CR)

or

LOAD“PRG”,8,1 (+SHIFT/RUNSTOP)

or

LOAD“PRG”,8,1 (+CR) SYS12345 (+CR)

Since typing is usually not the kind of work one does enjoy when working with a computer (especially with those sticky breadbin-keyboards) and sys-adresses are not always easy to remember, people found methods to make programs execute themselves during or directly after the load process. A side effect of this is that the instant execution makes it at least for learner-crackers a bit more difficult to examine the code (and then locate and disable copy protection routines, for instance).

Up to this point my research dug out 4 different methods to achieve self-booting abilities. While the first two were developed by other people, I haven't seen the last two except in my own programs (still I doubt I could claim to have 'invented' them).

Each of these methods exploits some of the BASIC/KERNAL functions that take place when loading a prg like shown above in one way or another. In a simplified form this is what happens after various interpreter routines found out what you wanted with that “load…”-stuff:

1) The kernal load-routine will set up filename, devices etc. according to what was found in the user-input.

2) The “load-byte”-loop inside that routine will read the file byte after byte, and if neither the end of file is reached nor a load-error occurs store the bytes in memory at the address stored in $ae/$af. After each received byte that pointer is incremented (and thus points to endaddr+1 after loading), and it is checked if the user pressed run/stop to terminate loading.

3) After the file has been loaded successfully the load-routine quits, “READY.” is printed and the basic interpreter is back at idle state to await new user inputs.

4) During all this the standard interrupt handler gets active every 1/60th of a second to scan the keyboard, update the clock etc pp.

Any but the 1st step can be used to get the boot-code activated (since the part of the file containing it must have been loaded before). To be automagically executed the boot code must be 'mistakenly' activated by the kernal routines, and therefore must be loaded to a memory area where it can bend some of the indirect jump-vectors used by the kernal to its start address.

Since all those vectors are words, unexpected stuff would happen if only the low or hi byte had been changed when the vector is used. The following examples take care of this.

METHOD #1: RTS TO STACK

This method requires a short stand-alone boot-code that loads into the stack memory from $102-$1ff and then loads and starts another file containing the main code. I forgot where I found this method, but believe to remember that it was used in a similar form in a copy of Friday the 13th I once had. The source-code below is Taboo TASS compatible and works like this:

```
* = $102    ;this MUST BE the autostart address

    lda #$7f    ;suppress any irq&nmi
    sta $dc0d    ;to disallow abort load by R/S
    sta $dc0e
    jsr $ff8a    ;restore i/o-vex (just for sure...)

    lda #<msg    ;print funky message (optional)
    ldy #>msg
    jsr $ab1e

    lda #namelen
    ldx #<name
    ldy #>name
    jsr $ffbd    ;set name of file to load

    lda #8
    tax
    tay
    jsr $ffba    ;open8,8,8

    lda #0    ;load... (lda #1 would verify)
    sta $9d    ;flag progam mode (to suppress 'searching for...' msg)
    jsr $ffd5    ;...filename,8,8

    jmp execaddr    ;start it

msg  .text "autoboot demo by tfg",0   ;enter your (short!) text here
name .text "superprg"     ;enter name of prg-to-load here
namelen = *-name
execaddr    = 2064    ;set to the 1st instruction's address of prg-to-load

;now comes the smart part:

.rept $200-*    ;this will add $01s from current address of loader to $1ff
  .byte1
.next
```

Loading the remainder of stack with $01s is the trick that does the job: If the booter-load has finished and control should be returned to the basic interpreter, sooner or later an RTS instruction is executed that will find the stack reading $0101. Since the RTS instruction adds 1 to the lb/hb address fetched from stack, it will now 'return' to $0102, where the boot-code awaits. Nice, isn't it?

The main drawback is that the prg-to-boot has to be in another file that could be loaded seperately. If you use this method to suppress examination of the code you'd had to encrypt that file (with a simple

eor-encryption or something alike) and make decryption depend on some value inside the loader code.

Also make sure that the message text is not that long that there is not enough memory left for the $01 fill-bytes.

METHOD #2: STOP-WEDGE

This method exploits that the loading-loop checks for the stop key after every byte and therefore the boot-code loads to the small area of unused ram below the system's jmp (indirect) vectors and overwrites the check-stop-key-vector. I derived the routine below from what I found in the Last Ninja 2 Music Demo by STA:

```
* = $02ed   ;DO NOT CHANGE, else the autostart will fail

    lda #$f6    ;repair stop check vector right away (only the
    sta $329    ;hi-byte was altered, that's why *=$02ed)
    lda #<$xyz    ;change end of file to start adress
    sta $ae    ;of the main code (e.g. $801 or $c000)
    lda #>$xyz
    sta $af
    jsr $f4f3    ;call load-code at get_next_byte-loop start

    jmp execaddr    ;start main code


;the system vectors at $300-$327 must remain intact to allow normal
basic/kernal
;operation and therefore the loader must contain the proper bytes for these:

* = $300    ;the vector table for basic/kernal

.word $e38b ;$300 vector: print basic error message ($e38b)
.word $a483 ;$302 vector: basic warm start ($a483)
.word $a57c ;$304 vector: tokenize basic text ($a57c)
.word $a71a ;$306 vector: basic text list ($a71a)
.word $a7e4 ;$308 vector: basic char. dispatch ($a7e4)
.word $ae86 ;$30a vector: basic token evaluation ($ae86)
.byte 0,0,0,0   ;$30c temp storage cpu registers

jmp $b248   ;$310 usr function, jmp+address
.byte 0 ;$313 unused

.word $ea31 ;$314 Vector: Hardware Interrupt ($ea31)
.word $fe66 ;$316 Vector: BRK Instr. Interrupt ($fe66)
.word $fe47 ;$318 Vector: Non-Maskable Interrupt ($fe47)
.word $f34a ;$31a kernal open routine vector ($f34a)
.word $f291 ;$31c kernal close routine vector ($f291)
.word $f20e ;$31e kernal chkin routine ($f20e)
.word $f250 ;$320 kernal chkout routine ($f250)
.word $f333 ;$322 kernal clrchn routine vector ($f333)
```

```
.word $f157 ;$324 kernal chrin routine ($f157)
.word $f1ca ;$326 kernal chrout routine ($f1ca)


;HERE'S THE TRAP:
.word $02ed ;$328: kernal stop routine Vector ($f6ed)

;IF YOU DON'T RELOCATE THE LOAD ADDRESS, ADD THE DATA BELOW TO THE CODE!
;.word $f13e    ;$32a kernal getin routine ($f13e)
;.word $f32f    ;$32c kernal clall routine vector ($f32f)
;.word $fe66    ;$32e user-defined vector ($fe66)
;.word $f4a5    ;$330 kernal load routine ($f4a5)
;.word $f5ed    ;$332 kernal save routine ($f5ed)
;end of vectors

.binary something ;append the main-prg here
```

Unlike method1 this method allows to store everything in one file. As soon as the file has been loaded up to $329 the inevitable check for the stop key will activate the boot-code that in turn will alter the load-to-address at $ae/af to the start address of the prg-to-boot which begins with the next upcoming byte.

Note that the (re-)call of the loading-loop from within the boot-code will cause a slight stack mess-up. If you were to boot a routine that runs in the background on an irq you must throw away one return address at its start (pla, pla).

Also note that CCS64 will not execute this boot-method properly unless the file is part of a disk-image.

METHOD #3: IRQ-WEDGE

This method bends the irq-vector to autostart a custom irq-routine. It is the dirtiest autostart method I could think of (it overwrites quite some system values) and only works with relatively small prgs (like viruses, for instance ;)) The code below shows how I used this method to autostart a simple raster-effect:

```
* = $028f   ;DO NOT CHANGE, else the autostart will fail

.word setupirq  ;keyboard setup vector will point to irq-setup routine after
loading these
.byte 0,0,0,0   ;fillbytes to skip some randomly changing bytes

setupirq = *    ;this is done on the first irq after loading is done:

    lda #$7f
    sta $dc0d    ;disable CIA irq

    lda #<wedge    ;change irq-veccy to our final irq-wedge
    sta $314
    lda #>wedge
    sta $315
```

```
    lda #$48    ;IMPORTANT: restore old key-decode vector
    sta $28f
    lda #$eb    ;or any keypress will crash the machine!
    sta $290

    lda #$1b    ;set up raster compare at line $030 (#48)
    sta $d011    ;(just above visible screen)
    lda #$30
    sta $d012

    lda #1
    sta $d01a    ;enable VIC-irq


wedge =*        ;our short and dirty little effect :)

    lda $02    ;a free ZP-address, used as offset-storage

loop    ldy xoffset,x    ;the first pass is always a random one for X isn't
set
    sty $d016    ;change the VICs horizontal pixel-offset

    adc #$01    ;INC offset
    and #$1f    ;cycle through our 32 byte table
    tax    ;use as index

    ldy $d012    ;a little timing: get raster#
sync    cpy $d012    ;and wait until it is over
    beq sync

    cpy #$f8    ;loop until we reach end of visible screen, this
    bcc loop    ;clears the carry for the ADC #1 above :)

    inc $02    ;increase starting offset for the next frame

    inc $d019    ;acknowledge irq
    jmp $ea31    ;the standard irq-routine

xoffset     .byte $8,$8,$8,$8,$9,$9,$9,$a    ;some offsets for a wave-effect
    .byte $a,$b,$c,$d,$d,$e,$e,$e
    .byte $f,$f,$f,$f,$e,$e,$e,$d
    .byte $d,$c,$b,$a,$a,$9,$9,$9


;the tricky part. this is how the memory between our code & irq-vector
;looks like after power up.

* = $300    ;the vector table for basic/kernal is as follows:

    .word $e38b    ;$300 Vector: Print BASIC Error Message
    .word $a483    ;$302 Vector: BASIC Warm Start
```

```
    .word $a57c    ;$304 Vector: Tokenize BASIC Text
    .word $a71a    ;$306 Vector: BASIC Text LIST
    .word $a7e4    ;$308 Vector: BASIC Char. Dispatch
    .word $ae86    ;$30A Vector: BASIC Token Evaluation
    .byte 0,0,0,0    ;$30C temp storage cpu registers
    jmp $b248    ;$310 usr function, jmp+address
    .byte 0    ;$313 unused

    .word $eadd    ;$314/$315: here the irq will find the way to our routine
```

When the last bytes are transferred to memory the irq vector at $314/15 will be changed from $ea31 to $eadd. This was the only location I could find with an indirect jump on that ea-page. The jump vector at $28f has been altered during load as well, and thus will execute the raster-setup code that again changes the irq-vector to the main effect routine.

Unlike the previous method this one works with CCS64 and standalone .PRG-files, but unlike the previous method it is also pretty useless :)

METHOD #4: 'READY.'-TRAP

This is a short one that bends the vector at $326 that is part of the infamous $ffd2 charout routine. I usually use it in conjunction with a slightly modified version of the taboo decrusher. As soon as the load has finished and basic tries to print the ready-msg the autostart code is activated:

```
* = $326    ;DO NOT CHANGE, else the autostart will fail

.word boot  ;autostart from charout vector ($f1ca)
.word $f6ed ;$328 kernal stop routine Vector ($f6ed)
.word $f13e ;$32a kernal getin routine ($f13e)
.word $f32f ;$32c kernal clall routine vector ($f32f)
.word $fe66 ;$32e user-defined vector ($fe66)
.word $f4a5 ;$330 kernal load routine ($f4a5)
.word $f5ed ;$332 kernal save routine ($f5ed)

;* = $334 (cassette buffer)

boot    sei
    lda #$ca    ;repair charout vector ($f1ca)
    sta $326
    lda #$f1
    sta $327

    ...    ;decruncher routine
.binary crushed.bin ;crunched file
```

The above method should work very well with any kind of decruncher that works from the cassette-buffer. Since decrunchers kick around the crunched/decrunched data in memory anyway, there's no need to relocate the load-to-address of the main-code, and as a nice side-effect you can watch how the screen fills with garbage as the load progresses.

As a not so nice side effect this method will most of the time cause a crash if the load procedure is

interrupted by the stop-key or some other load-errors because the inevitably following screen-printout would start the decrunch process.

Finally, this method works with emulators like CCS64 whether the file is part of a disk image or not.

METHOD #5: Combination of STOP-WEDGE and 'READY.'-TRAP

by Oliver Jones, derived by myself.

This method is a more “natural” evolution of the STOP-WEDGE, as detailed above. This method has the advantage of not upsetting the stack, since it uses the 'READY.'-trap to actually run the code when it has finished loading, rather than hi-jacking the load process. (Of course, you need to reset the vector back to $f1ca when your main code is run - but that is simple enough.) I personally like this type of autoboot code best, as it means you can cleanly load your code anywhere between $0328-$9fff and $c000-cfff, without having to see rubbish load over the screen area ($0400-07e8). It also means you can do certain initialisation steps, such as clearing memory (very useful when used in combination with method 6, for example) - or setting up the display.

By the way, a start address of $02dc will also work, as it corresponds to an RTS instruction in ROM ($f6dc) - and this is the case for both the normal stock Commodore ROMS and JiffyDOS ROMs.

For the slightly more adventurous, it is also possible to implement funky loading schemes (like a self-decompressing loader) - this is easier than you might think, as there are three spare bytes - just enough for, say, a JSR $02a7 call inserted before the JMP $f6ed instruction (which would replace the three spare bytes) - and Robert's your father's brother. It is possible to initialise a counter, set the STOP vector and use a piece of code to “watch” how many bytes have been loaded in, then reset $ae/$af - allowing the chaining of multiple files in one executable. It's a little finicky, but it works.

However, if you feel really hardcore, you may be VERY interested to know that it is possible to completely hi-jack the loading process and micro-manage everything yourself: IECIN ($ffa5) will quite happily feed you the next byte in the file if you call it directly. (In this case, you need not bother with trapping the READY. prompt, since you can jump directly to the start address once your code has finished its work.) Of course, you will need to manage conditions like EOF yourself, but - as any user of sudo will tell you - with increased power also comes increased responsibility.

```
* = $02ed       ; $02dc will also work, but in that case you need to restore
$328 as well.

    lda #$f6    ;repair stop check vector right away (only the
    sta $329        ;hi-byte was altered, that's why *=$02ed)
    lda #<$xyz    ;change end of file to start adress
    sta $ae        ;of the main code (e.g. $801 or $c000)
    lda #>$xyz
    sta $af
    jmp $f6ed    ;surrender control back to normal loading routine

    .byte 0
    .byte 0
    .byte 0        ;three unused bytes ... could be useful!

;the system vectors at $300-$327 must remain intact to allow normal
```

```
basic/kernal
;operation and therefore the loader must contain the proper bytes for these:

* = $300    ;the vector table for basic/kernal

.word $e38b ;$300 vector: print basic error message ($e38b)
.word $a483 ;$302 vector: basic warm start ($a483)
.word $a57c ;$304 vector: tokenize basic text ($a57c)
.word $a71a ;$306 vector: basic text list ($a71a)
.word $a7e4 ;$308 vector: basic char. dispatch ($a7e4)
.word $ae86 ;$30a vector: basic token evaluation ($ae86)
.byte 0,0,0,0   ;$30c temp storage cpu registers

jmp $b248   ;$310 usr function, jmp+address
.byte 0 ;$313 unused

.word $ea31 ;$314 Vector: Hardware Interrupt ($ea31)
.word $fe66 ;$316 Vector: BRK Instr. Interrupt ($fe66)
.word $fe47 ;$318 Vector: Non-Maskable Interrupt ($fe47)
.word $f34a ;$31a kernal open routine vector ($f34a)
.word $f291 ;$31c kernal close routine vector ($f291)
.word $f20e ;$31e kernal chkin routine ($f20e)
.word $f250 ;$320 kernal chkout routine ($f250)
.word $f333 ;$322 kernal clrchn routine vector ($f333)
.word $f157 ;$324 kernal chrin routine ($f157)


;HERE'S THE TRAP:
.word $exec ;$326 kernal chrout routine ($f1ca)
.word $02ed ;$328: kernal stop routine Vector ($f6ed)

.binary something ;append the main-prg here
```

METHOD #6: $01-trap

by Oliver Jones, derived by myself.

This is one method that is sure to put hairs on your legs, quite simply down to its pure audacity. Here is how it works: You create a file that loads into RAM underneath $e000-$ffff (but, particularly, $fffa-$ffff) - and you set the IRQ and NMI vectors in the file (load position $fffa/$fffb and $fffe/$ffff) to the start of your code. However, you add two extra bytes - one for $00 and one for $01, at the end of the file - two bytes too many! Yes, you guessed it - when the $ae/$af pointers wrap past $ffff, they will go straight to $0000, and you can exploit this by mapping the kernal ROM out as the last byte of your file loads...

I'm afraid this method is not for the faint-hearted: You need to be aware that your code will be started by a BRK-interrupt, and the best way for that to happen is for you to clear the RAM from $e000-$ffff with zeros (probably as part of a loader initialisation routine - methods involving the STOP trap are good candidates.) When the ROM drops out from underneath the loader routine - be it JiffyDOS, Dolphin DOS, normal Commodore DOS - whatever - it will hit a BRK instruction and trap to the start of your code - which, hopefully, knows how to cope with interrupts.

Have fun. :)

METHOD #7: hybrid boot code

by Felix Palmen, derived from METHOD #5 (Oliver Jones)

Wouldn't it be nice for a user not having to care about HOW to load a program? If you're not concerned about protecting your code but just about ease of use, try this. You will have to sacrifice some bytes, but as a result, the user can do a LOAD“*”,8 and RUN as well as a LOAD“*”,8,1 to get autostart.

The trick is to use METHOD #5 and prepend a BASIC header (SYS xxxx), then choose the real load address so that it's directly after $0801 + [BASIC header] + [autostart code]. The following example uses cc65/ca65 syntax:

```
.segment "BOOT"

        .word    $02e0        ; load address for binary
        .assert    *=$02e0, error    ; check linker placed us correctly

; BASIC header -- must be here if loaded with ",8" only
basichdr:       .word   $080b
                .word   $17     ; 23
                .byte   $9E     ; SYS
                .byte   "2123", 0
                .byte   0
                      .word   0
basichdrlen = *-basichdr

; entry of hijacked STOP routine
        .assert *=$02ed, error
        lda    #$f6    ;repair stop check vector right away (only the
        sta    $329    ;hi-byte was altered, that's why *=$02ed)
        lda    #<loader    ;change end of file to start adress
        sta    $ae        ;of the main code
        lda    #>loader
        sta    $af
        jmp    $f6ed    ;jump back to normal loading routine
  .byte 0,0,0

;the system vectors at $300-$327 must remain intact to allow normal
basic/kernal
;operation and therefore the loader must contain the proper bytes for these:

        .assert *=$0300, error    ;the vector table for basic/kernal
        .word $e38b    ;$300 vector: print basic error message ($e38b)
        .word $a483    ;$302 vector: basic warm start ($a483)
        .word $a57c    ;$304 vector: tokenize basic text ($a57c)
        .word $a71a    ;$306 vector: basic text list ($a71a)
        .word $a7e4    ;$308 vector: basic char. dispatch ($a7e4)
        .word $ae86    ;$30a vector: basic token evaluation ($ae86)
        .byte 0,0,0,0    ;$30c temp storage cpu registers
```

```
        jmp $b248    ;$310 usr function, jmp+address
        .byte 0        ;$313 unused

        .word $ea31    ;$314 Vector: Hardware Interrupt ($ea31)
        .word $fe66    ;$316 Vector: BRK Instr. Interrupt ($fe66)
        .word $fe47    ;$318 Vector: Non-Maskable Interrupt ($fe47)
        .word $f34a    ;$31a kernal open routine vector ($f34a)
        .word $f291    ;$31c kernal close routine vector ($f291)
        .word $f20e    ;$31e kernal chkin routine ($f20e)
        .word $f250    ;$320 kernal chkout routine ($f250)
        .word $f333    ;$322 kernal clrchn routine vector ($f333)
        .word $f157    ;$324 kernal chrin routine ($f157)

        ;HERE'S THE TRAP:
        .word loader    ;$326 kernal chrout routine ($f1ca)
        .word $02ed    ;$328: kernal stop routine Vector ($f6ed)

.segment "CODE"

loader:
        .assert    *=$084b, error    ; check linker placed us correctly
        lda    #$f1
        cmp    $327
        beq    normalstart    ; didn't autostart
        sta    $327
        lda    #$ca
        sta    $326
        pla             ; if loaded from autostart, throw away
        pla             ; last return address (from CHROUT call)
normalstart:
  [...]
```

This shold work with the following ld65 linker config:

```
MEMORY {
    BOOT:   start = $02de, size = $a000-$084b;
    CODE:   start = $084b, size = $a000-$084b;
}
SEGMENTS {
    BOOT:   load = BOOT;
    CODE:   load = BOOT, run = CODE;
}
```

The start of BOOT memory is 2 less than the actual load address (02e0) to make room for the load address in the output binary. All this code is UNTESTED, so please forgive me if some addresses are miscalculated – I just hope you get the idea!

Of course you can put some code between the basic header and the STOP trap code, as described in METHOD #5. Just make sure you adjust the BOOT load address as well as the base for “loader” (CODE segment) and the decimal representation in the BASIC header. I have a working real-life example here: https://github.com/Zirias/c64_demo_part_zirias/blob/master/kickstart.s