# SuperBoot v1.00
## for SuperCPU
**Revised Edition (4/17/2017)**

## SuperCPU Initialization and Payload Launch Vehicle

**Create large, multi-bank SuperCPU programs in read-only memory. Boot automatically.**

**Original Article by**
**Bert Novilla (satpro)**
**11/20/2016**

Note: Emulation is currently the only avenue for activities of this sort. Thankfully, SuperCPU emulation in VICE is excellent. But, just for the record... ***actual hardware would be more excellent***.

### Auto-boot a SuperCPU?
In this article, I will show how to create the program that starts and controls a SuperCPU. This program is a block of code commonly referred to as a boot or BIOS. What is a boot? For our purposes, a boot is "*hardware initialization code in read-only memory that passes control to a secondary, payload program in RAM.*" The payload can be anything. During computer initialization, the "boot" sets things up and then hands off to the payload. In this article, we will build a SuperCPU boot from scratch.

Our boot program, which we will call SuperBoot, requires neither C64 Kernal nor BASIC in order to function. Rather, it assumes neither is present. In a way, SuperBoot is a payload-assist service. It initializes computer hardware and helps launch a secondary program, aka the payload. SuperBoot starts at computer power-on, when the CPU is in Emulation mode and retrieving the RESET vector. Following this, SuperBoot implements a set of distinct steps, leading to an eventual handoff to some other control program. I use a version of this template to launch my experimental 16-bit operating system, Winc64. Without SuperBoot, I would have to load Winc64 every time I wanted to use it.

### Why SuperBoot?
SuperBoot is very small and blindingly fast. It starts in VICE similar to a cartridge, automatically, except that the secondary program it delivers, the payload, will be running at a full 20 MHz in fast static RAM. This is akin to using a CMD or VICE image, except their payloads, C64 Kernal and BASIC, run in Emulation mode and thereafter, must load everything from disk or cartridge. If a program does switch to Native mode, hidden SuperCPU interrupt handlers switch to Emulation mode and back *every time an interrupt occurs*. To make matters worse, SuperCPU cartridges operate at 1 MHz, and the only cartridges that work are of the 8K variety.

With all of this in mind, I decided to write SuperBoot.

### Design Choices.
SuperBoot is minimalist. Working in 65816 Native mode, it initializes a 65816, SuperCPU hardware, and C64. It blanks the screen and border to black and passes control to a secondary, sample payload program at location $1000 in Bank $00. The rest is up to the payload program.

By keeping it small and simple, we eliminate many headaches. In reality, a full-featured BIOS would do much more. Most importantly, the CPU is in Native mode and the C64 is set for all RAM, so both 65816 and SuperCPU behave by the book, and all Emulation mode complications and quirks vanish. The payload SuperBoot delivers will not have access to a Kernal or BASIC in C64 ROM. It will be in an environment we refer to as "bare metal" and this means *there is no safety net*. One could add BASIC or Kernal as additional payload programs, but to SuperBoot and SuperCPU, *they do not yet exist*.

## Ready. Reset. Go.

At power-on, a set of instructions executes, first in RAM Bank $00, then in ROM Bank $F8. This occurs *millions of cycles before* the C64 Kernal accepts control and shows the blinking "READY" prompt. Chances are, most readers have never seen the first instructions that execute in a SuperCPU. This is immediately after power-on, in read-only memory located in Bank $F8. Before that there is RESET, the point in time when a 65816 is *reset electrically* and begins executing instructions. RESET is the starting point for everything that happens next. As it turns out, we can specify *what happens next*.

## Native mode, SuperCPU, and its registers.

A 65816 CPU performs operations on 16-bit data and directly accesses a 24-bit address space. SuperBoot places the payload in fast static RAM (banks $00-$01), which is memory that runs at the full 20 MHz. Incidentally, a surprisingly small number of hardware registers will control a SuperCPU effectively.

To aid in learning how a SuperCPU works, I studied the original binary image from CMD as well as source code for the VICE emulator. I also obtained unpublished VICE boot ROM source, and it was invaluable. While not plentiful, there was enough information online to gather a reasonably accurate picture of how a SuperCPU works. I have also documented dozens of other locations for RAMLink and RLDOS, but they are not important here and are not referenced further.

## CMD 2.04

The CMD 2.04 image (128K) is elaborate and contains an enormous amount of code tasked with proprietary device communication, among many, many other things. A zillion C128 machine checks litter the startup code. The 2.04 binary includes the famous "CMD Space Rocket" splash animation, JiffyDOS, RAMLink, and for the purposes of the VICE emulator, a useless second bank which is devoted to the C128. It should be noted that the C64 Kernal is heavily patched for use in the SuperCPU.

## VICE 0.05-0.06

In contrast, the VICE 0.05 image (64K) consists of minimal code devoted primarily to controlling a SuperCPU and dealing with interrupts in 8-bit Emulation mode. VICE does not worry about a C128, RAMLink, or JiffyDOS, but just like CMD, performs a memory test and confines itself to Emulation mode.

**SuperBoot v1.00** boots a SuperCPU 76% faster than VICE, and 99.9% faster than CMD. The secret is... Native mode.

## Power-on and Bootmap mode.

At power-on, a 65816 is in 8-bit Emulation mode. Instruction execution commences at the 16-bit address pointed to by locations $FFFC-$FFFD, in Bank $00 -- the normal RESET vector everyone knows. Normally this address is in ROM, but the SuperCPU contains RAM in Bank $00. There is no ROM. To overcome this issue, the SuperCPU contains an operational environment known as *Bootmap mode*.

In Bootmap mode, the CPU "sees" the upper 32K bytes of *read-only* memory (ROM) Bank $F8 instead of random-access memory (RAM) in Bank $00, excluding the 4K I/O block at $D000. With clever address mapping, Bootmap mode fools a SuperCPU into seeing ROM from Bank $F8 where RAM exists in Bank $00.

**Note:** VICE requires ROM images to contain 64K, 128K, 256K, or 512K bytes. 64K segments are numbered 0 - 7, and SuperCPU reserves banks $F8-$FF for read-only memory. In this article, read-only memory in Bank $F8 is referenced as "ROM Bank $F8".
The power-on indirect vector landing point, which can be as simple as a long jump back to ROM Bank $F8, is located within the second 32K block of memory in ROM Bank $F8 (excluding the $D000 I/O block). While Bootmap mode is active, a 65816 thinks it is working in actual Bank $00 memory. *This is how a SuperCPU finds the non-existent RESET vector at $FFFC*. Just after RESET via $FFFC, SuperBoot lands at $FFE0 -- right into the loving arms of a long jump (JML) back to ROM address $F8:0000.

With CMD and VICE images, a SuperCPU is a 16-bit machine forced into compatibility with the limited 8-bit environment of a C64, Kernal, and BASIC. An operating system must allow programs to run in Native mode, yet both the Kernal and BASIC run assuming a 6510. Without modification, the stock C64 Kernal will not function properly in Native mode running at 20 MHz. There are several additional SuperCPU hardware issues with an

Emulation mode kernal, as noted elsewhere in this document. SuperBoot's approach is to activate Native mode at 20 MHz and stay there, and to assume a C64 Kernal does not exist. If desired, one could modify and integrate a C64 Kernal and BASIC into the payload mix, as CMD and VICE do, but a it is *not* required for SuperCPU operation.

Later on, Bootmap mode is disabled and normal mapping resumes as initialization continues. The final task in ROM Bank $F8 is to pass control to the payload program. Arbitrarily, I chose location $1000 in Bank $00 as the handoff address, but it could be *any address* in Banks $00-$01, with a few exceptions. At any rate, a new control program will take over from there. SuperBoot is finished.

## *"Inside SuperBoot"*

At power-on, a SuperCPU is in Bootmap mode, a 65816 is in Emulation mode, and a C64 is in RESET. After a long jump from $FFE0 in Bank $00, hardware initialization *begins* in ROM Bank $F8, but does not remain there for very long. Read-only memory is slow, relatively speaking -- barely half as fast as static random-access memory -- so only important tasks are performed by instructions in ROM Bank $F8. Everything else takes place later at full-speed in fast static RAM Bank $00.

Below is an example showing the last 192 bytes of ROM Bank $F8. This block is largely unused, yet vital to the boot process. Native mode interrupt vectors at $FFE4-$FFEF behave just as they do with a C64, but are in a different place. Notice the long jump at $FFE0. This is a four-byte springboard to ROM Bank $F8, the place where hardware initialization begins. *($FFE0 is SuperBoot's "$FCE2" equivalent)* SuperBoot is not concerned with any of the interrupt vectors, and leaves interrupts disabled. The payload will determine and configure interrupts.

In the table below, the *mostly empty* 192-byte block shown is visible in Bank $00 only while Bootmap mode is active, so SuperBoot copies this block from ROM Bank $F8 to RAM Bank $00 in order to be visible at all other times. The 160-character copyright message block space, located at $FF40 and extending to $FFDF, is cosmetic and optional. VIC Bank 3 bitmap graphics will not affect the copyright message because it is out of range.

### Source Code Format.

In all code samples, there are no symbolic names, labels, or macros. The source text is a pseudo-monitor listing, designed to be easily understood.

**Vector Table, Springboard, and Copyright Message at $F8:FF40**
**Mapped to $00:FF40 when Bootmap mode is active**

```
*= $ff40                ;your copyright message could go here (max. 160 bytes)

*= $ffe0                ;"Reset"    5C 00 00 F8

      jml $f80000       ;springboard: long jump to ROM at $f8:0000

*= $fffc                ;RESET      E0 FF
.word $ffe0             ;RESET points to "Reset" ($ffe0)
```

### 65816 Startup

First, SuperBoot initializes the 65816 CPU. Some values, such as Stack Pointer and Direct Page, are arbitrary and default. The payload program may change the values anyway. The Data Bank register is set to $00, which allows SuperBoot to use normal addressing in RAM Bank $00 -- *while executing instructions in ROM Bank $F8*. SuperBoot will remain in Native mode until handoff, but ultimately, the payload determines what happens after SuperBoot has finished. In reality, a robust boot program would anticipate a rogue-entry situation, e.g. a warm start. For simplicity, SuperBoot assumes the only entry point is through hardware reset.

For this first code fragment, the CPU operates with a 16-bit accumulator and 8-bit index registers. As stated earlier, interrupts will remain disabled until handoff.

**65816 Initialization at $F8:0000 (Start)**

```
*= $f80000              ;$F80000 in bank rom $f8

;SuperBoot lands here after a long jump from $00ffe0...

;*** 65816 is in Emulation mode, SuperCPU is in Bootmap mode ***

;----- 65816 CPU NATIVE MODE INITIALIZATION -----

78          sei         ;disable interrupts

18          clc
FB          xce         ;enter Native mode AM = 8-bits, XY = 8-bits

C2 20       rep #$20    ;set 16-bit AM

A9 FF 01    lda #$01ff
1B          tcs         ;Stack Pointer (SP) = $01ff

A9 00 00    lda #$0000
5B          tcd         ;Direct Page (DP) = $0000

AA          tax         ;copy 0 into 8-bit X
DA          phx         ;do a "push-pull" of a 0 byte
AB          plb         ;Data Bank (DB) = $00, R/W data in Bank $00)
```

**65C816 Interrupt Vector Locations**

| Emulation Mode | | Native Mode | |
|---|---|---|---|
| Interrupt Vector | Location | Interrupt Vector | Location |
| IRQ/BRK | $FFFE,F | IRQ | $FFEE,F |
| RESET | $FFFC,D | (reserved) | $FFEC,D |
| NMI | $FFFA,B | NMI | $FFEA,B |
| ABORT | $FFF8,9 | ABORT | $FFE8,9 |
| (reserved) | $FFF6,7 | BRK | $FFE6,7 |
| COP | $FFF4,5 | COP | $FFE4,5 |

### Disable C64 hardware.

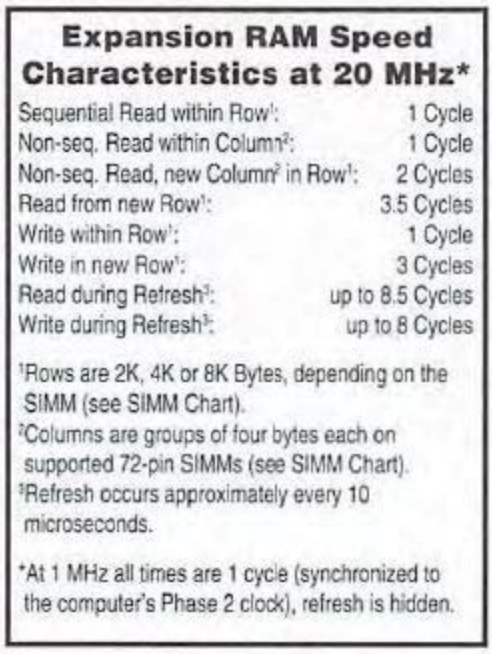
Now, it is time to visit a few key hardware registers in the C64 itself. Because read-only memory is slow memory, SuperBoot limits time-consuming activities while in ROM. For now, it will disable simulated C64 ROM memory, and blank the screen to a nice, clean black.

### Accessing C64 hardware.

For a SuperCPU, accessing C64 hardware is always expensive, even while operating in static RAM. In order to be 100% effective, C64 I/O accesses should be spaced about 20 CPU cycles apart. Anything less, on average, could result in a slight delay while waiting for the next 1 MHz I/O access cycle. In reality, one will probably

never have to worry about this, as there is a one-byte hardware buffer to help mitigate I/O mirroring wait states.

The SuperCPU runs at 20 MHz; the C64 hardware runs at 1 MHz. You either wait, or do something else for 19 20 MHz cycles while C64 I/O accesses take place. It really is this simple. The payload program should confine its processing activity to static RAM Banks $00-$01 as much as practicable. Static RAM *guarantees* 20 MHz. All other memory is slower, to varying degree. Period.



**Expansion RAM Speed Characteristics at 20 MHz***

| | |
|---|---|
| Sequential Read within Row[1]: | 1 Cycle |
| Non-seq. Read within Column[2]: | 1 Cycle |
| Non-seq. Read, new Column[2] in Row[1]: | 2 Cycles |
| Read from new Row[1]: | 3.5 Cycles |
| Write within Row[1]: | 1 Cycle |
| Write in new Row[1]: | 3 Cycles |
| Read during Refresh[3]: | up to 8.5 Cycles |
| Write during Refresh[3]: | up to 8 Cycles |

[1]Rows are 2K, 4K or 8K Bytes, depending on the SIMM (see SIMM Chart).
[2]Columns are groups of four bytes each on supported 72-pin SIMMs (see SIMM Chart).
[3]Refresh occurs approximately every 10 microseconds.

*At 1 MHz all times are 1 cycle (synchronized to the computer's Phase 2 clock), refresh is hidden.

**64K RAM, Disable C64 I/O Hardware**
**MMU, VIC, CIA, and SID**

```
A2 E5       ldx #$e5          ;MMU:  disable simulated ROM, enable I/O
86 01       stx $01           ;c64:  $a000/$e000 = RAM, $d000 = I/O
A2 2F       ldx #$2f          ;set MMU data direction to output, disable datasette
86 00       stx $00

A2 00       ldx #$00          ;border (clean, black boot screen)
8E 20 D0    stx $d020         ;VIC border register
11 D0       stx $d011         ;blank the screen output to border color

A2 7F       ldx #%01111111    ;#$7f   disable all cia interrupts
8E 0D DC    stx $dc0d         ;CIA #1 Interrupt Control Register
8E 0D DD    stx $dd0d         ;CIA #2 Interrupt Control Register

AD 0E DC    lda $dc0e         ;CIA #1 Control Register A
29 FE FE    and #$fefe        ;16-bit mask
8D 0E DC    sta $dc0e         ;cia #1 16-bit stop timers A & B

AD 0E DD    lda $dd0e         ;CIA #2 Control Register A
29 FE FE    and #$fefe        ;16-bit mask
8D 0E DD    sta $dd0e         ;cia #2 16-bit stop timers A & B

A2 1E       ldx #30           ;clear SID w/ 16-bit loop
9E 00 D4 -  stz $d400,x
CA          dex
CA          dex
10 F9       bpl -
```

### SuperCPU hardware

From here, SuperBoot moves on to SuperCPU hardware. To unlock write access, it performs a store (poke) to location $D07E. This activates SuperCPU hardware registers and permits write access. Storing to location $D07F de-activates the registers and resumes read-only status.

In Bank $00, the $D200-$D3FF block holds 512 bytes covered by hardware write-protection, accessed as described above. For example, CMD stores the amount of free RAM in this block. SuperBoot does not use any of this memory.

**The Free RAM Test and Checksum.**
One of the ways SuperBoot saves time is by eliminating the RAM test. This may have been necessary with real hardware, but in VICE we are *emulating* a 16MB SuperCPU. We know our RAM is good, and we know how much there is. There is no need for this extremely costly, long loop. For these reasons, SuperBoot does not perform a checksum calc.

Note: Bonus! We gain 128K, because CMD (and VICE) reserve RAM Banks $F6-F7 for "system use" and mark those banks as off-limits. Banks $F6-$F7 may be freely used.

20 MHz Turbo is selected, and memory optimization is turned off in preparation for the payload copy. Bootmap mode is disabled, and hardware registers are de-activated.

**Initialize SuperCPU Hardware**

```
;----- INITIALIZE SUPERCPU HARDWARE -----

8E 7E D0    stx $d07e         ;activate hardware registers

;--

A2 84       ldx #%10000100    ;#$84 payload will specify a vic bank later
8E B3 D0    stx $d0b3         ;= full speed (turns off all mirroring)

8E 7B D0    stx $d07b         ;software speed select = 20 mhz
8E 73 D0    stx $d073         ;system disable 1 mhz

A2 04       ldx #$04
8E 78 D0    stx $d078         ;set SIMM type (16 mb)

8E B6 D0    stx $d0b6         ;disable bootmap ($f8 mapped into $00 @ $8000-$ffff)

;--

8E 7F D0    stx $d07f         ;de-activate hardware registers
```

That is it. All that remains is to copy the payload program to RAM Bank $00.

**The Payload Copy.**
The possibility exists for some program to require use of all 512K ROM. For efficiency, a payload should place as much code and data as possible into RAM Banks $00 and $01. For that, SuperBoot copies memory from ROM Bank $F8 to RAM Bank $00, in one or more payload blocks. In our example here, SuperBoot delivers data to two different places in RAM Bank $00 using instruction MVN.

CPU registers A, X, and Y are set to 16 bits for the copy. For ease here, source and destination addresses are the same, but do not need to be. Remember to preserve the Data Bank register while executing a MVN instruction. From ROM to RAM, transfer rate is approximately 1 byte/us. (1MB/sec.). This is about one byte per C64 cycle.

**Deliver the Payload Program to RAM Bank $00**

```
C2 30       rep #$30          ;go16axy
8B          phb               ;save data bank because it is not retained

A9 FF 07    lda #$0800-1      ;size-1     copy some user-defined 2048-byte payload
A2 00 10    ldx #$1000        ;src addr
A0 00 10    ldy #$1000        ;dst addr
54 00 F8    mvn $f8, $00      ;copy 2048 bytes from $f8:1000 to $00:1000

A9 BF 00    lda #$00c0-1      ;size-1     copy the 192-byte vector table
A2 40 FF    ldx #$ff40        ;src addr
A0 40 FF    ldy #$ff40        ;dst addr
54 00 F8    mvn $f8, $00      ;copy 192 bytes from $f8:ff40 to $00:ff40

AB          plb               ;restore data bank
```

**Wrapping up.**
Earlier, SuperBoot disabled memory optimization. Memory optimization targets a VIC bank and only mirrors data from the specified 16K bank.

With that, the initialization routine is complete! The payload program at $00:1000 takes over and SuperBoot exits. At this point, a typical payload program might load its graphics or display a splash screen. SuperBoot however, is finished.

**The End**

```
;-----  done, exit to payload program coldstart  -----

5C 00 10    jml $001000 ;payload coldstart
```

**Some final thoughts.**
That is all there is to it. 124 bytes, 2325 1 MHz cycles -- with a 2K payload. Not bad, eh? The computer is now humming along at a full 20 MHz in the fastest type of RAM available. For this article, SuperBoot was limited intentionally. One idea is to expand this template with more features and options. For example, you could load settings from the REU, the contents of which can be saved in VICE until the next session a la RAMLink. You could also devise a 16-bit Kernal and BASIC (or some other scheme). For this article we have assumed the payload will manage its own operating environment, and purposely limited our initialization.

Enjoy.

For further information, please email me at bert@winc64.com or PM "satpro" on Lemon64.

Above: A screenshot from my experimental operating system, Winc64.

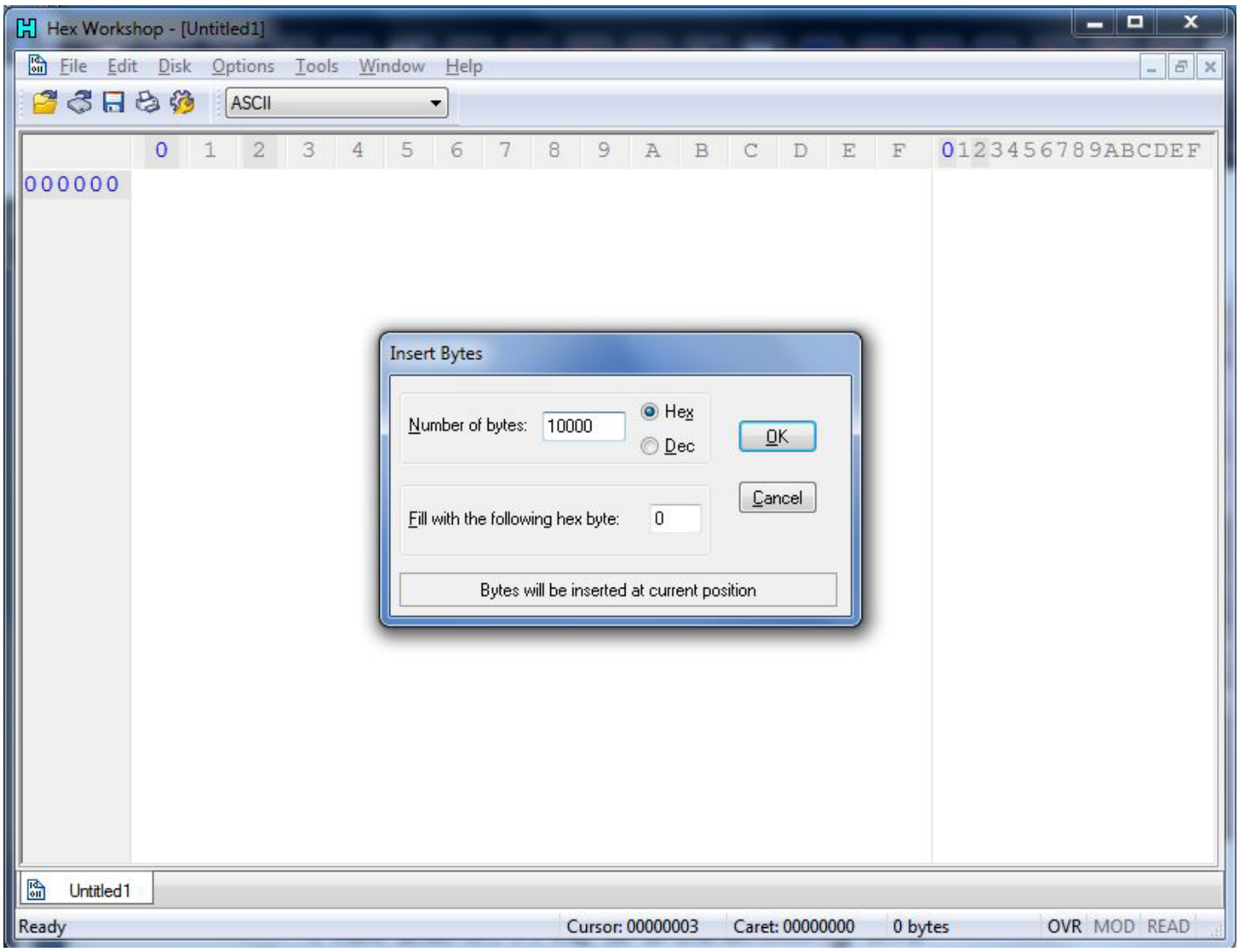
# SuperBoot v1.00

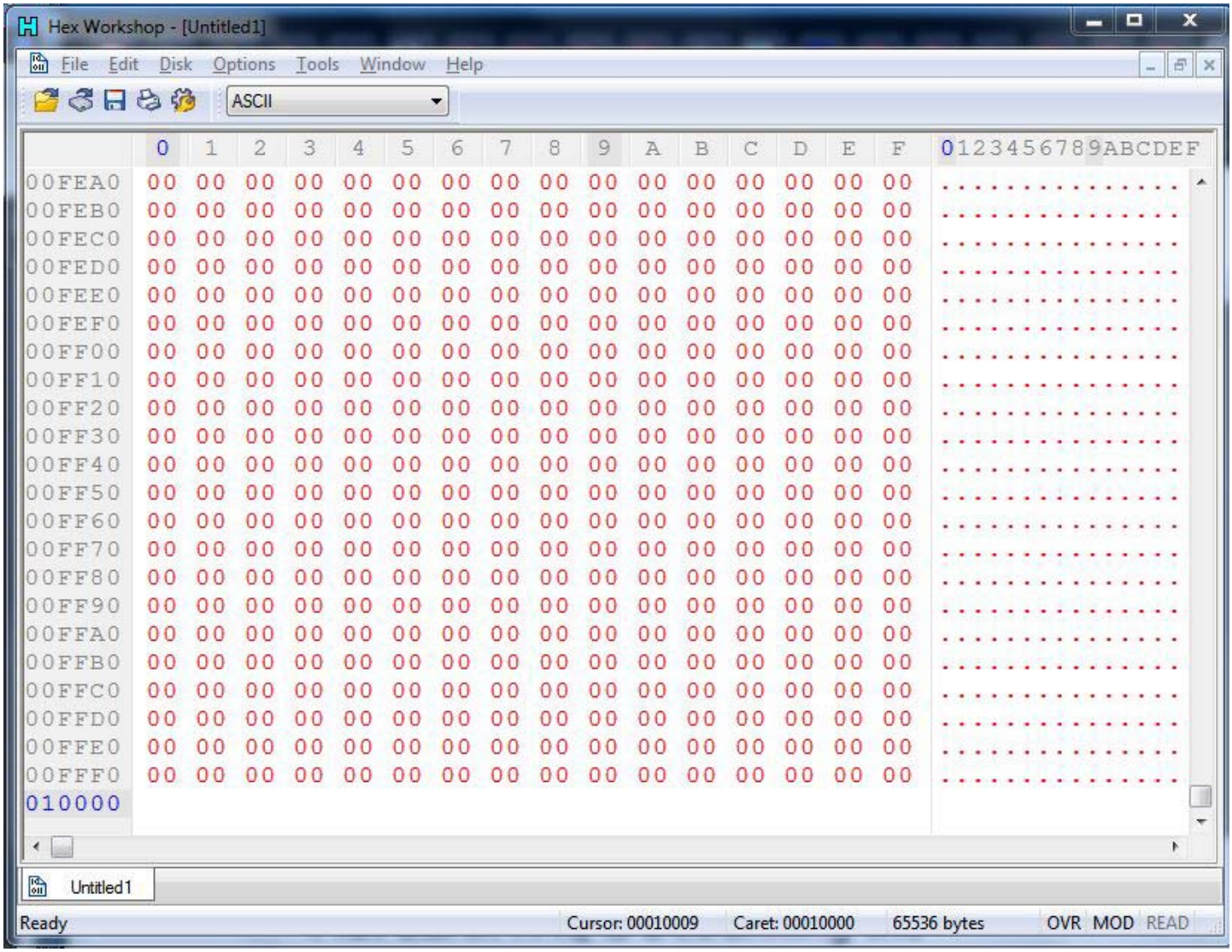
**for SuperCPU**

## How to Build the ROM Image

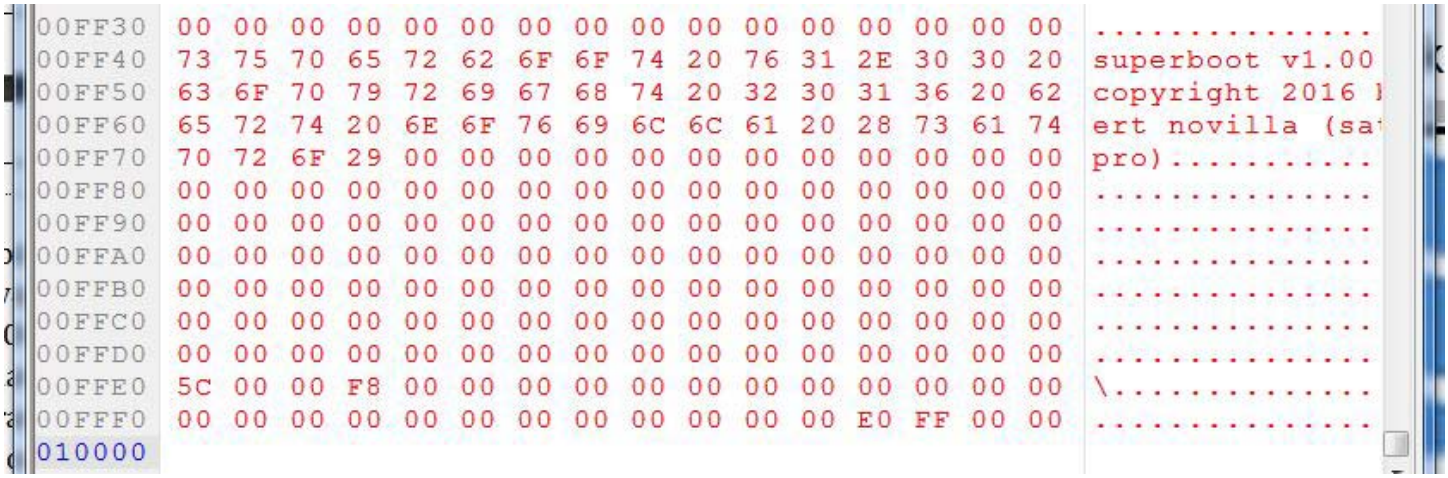
In order to build an image of SuperBoot, you will need a hex editor. For version 1.00, we will build a 64K image, but you may also try 128K, 256K, or 512K. Programs should download code and data to static RAM Banks $00-$01 prior to time-intensive operations involving the data.

1. Create a new 64K image, filled with 00 bytes. The image size is 65536 bytes.

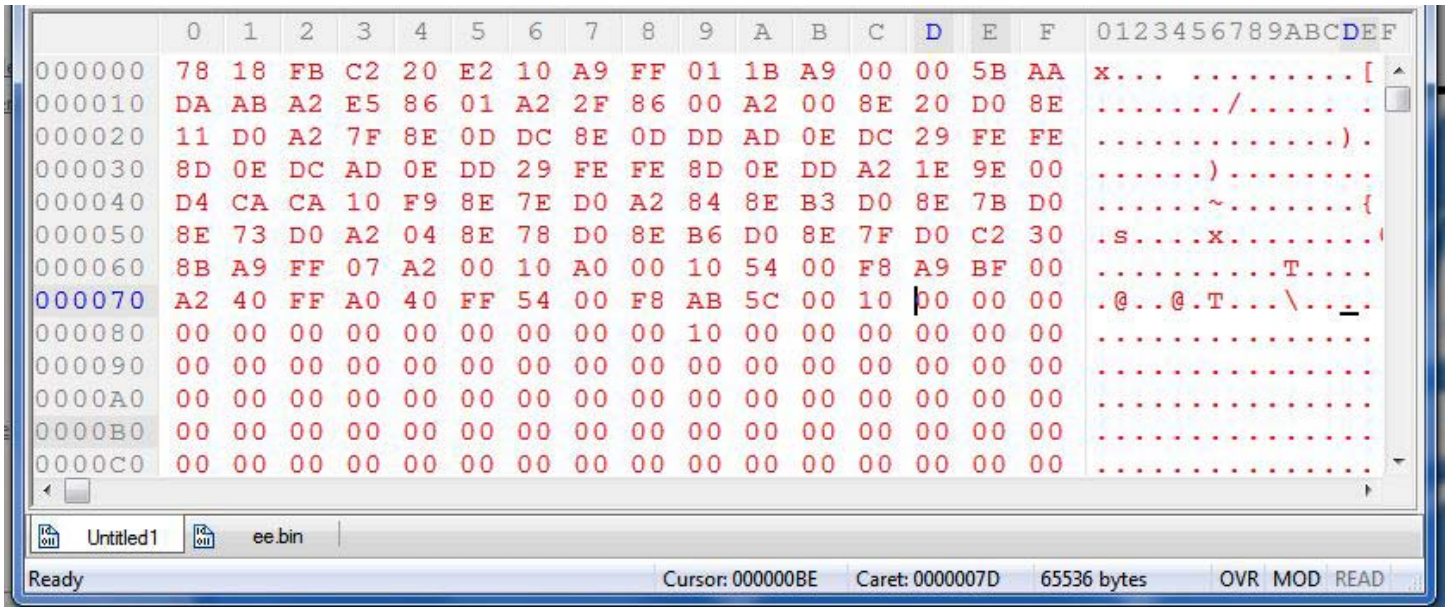




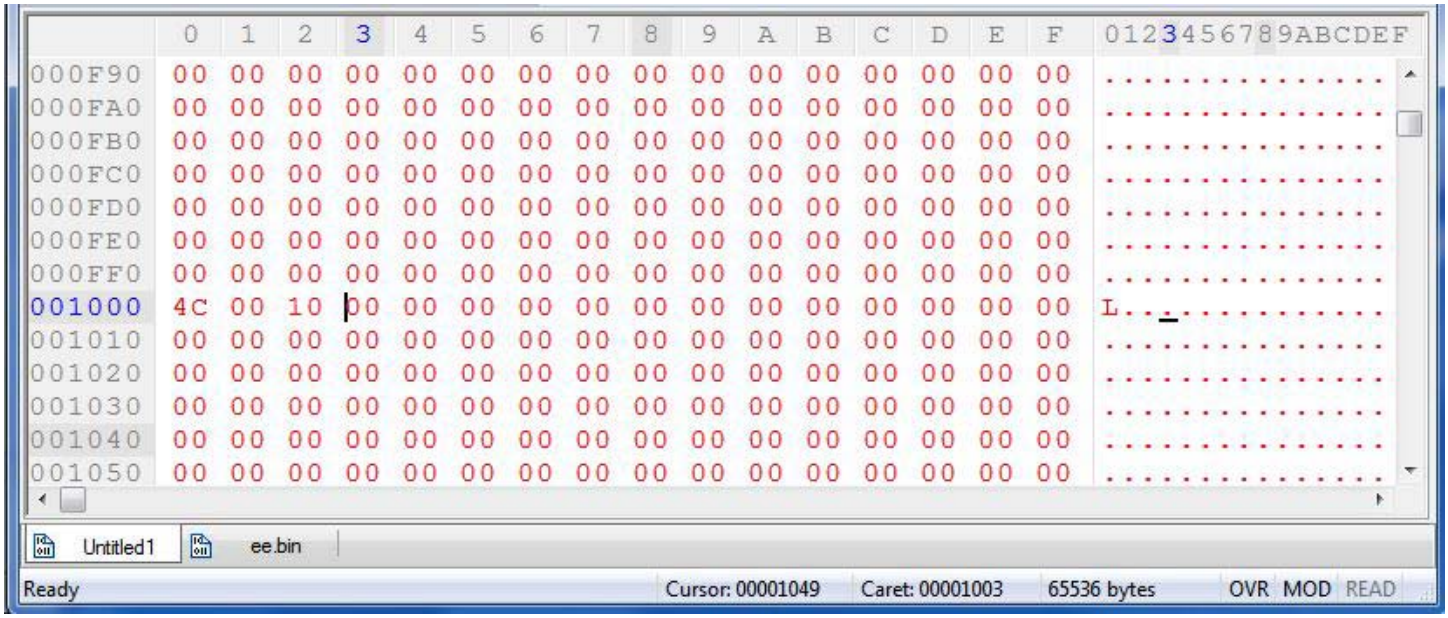

2. Insert: E0 FF at $FFFC. Insert: 5C 00 00 F8 at $FFE0. Copyright text at $FF40 is optional.



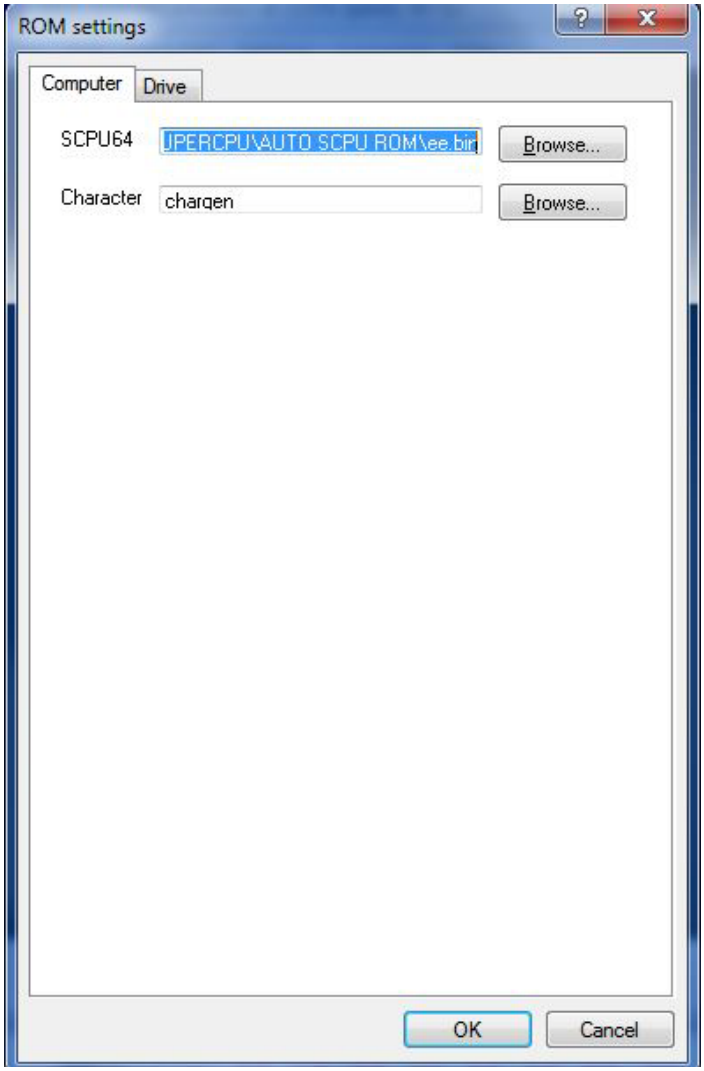
3. Insert SuperBoot binary at $0000.



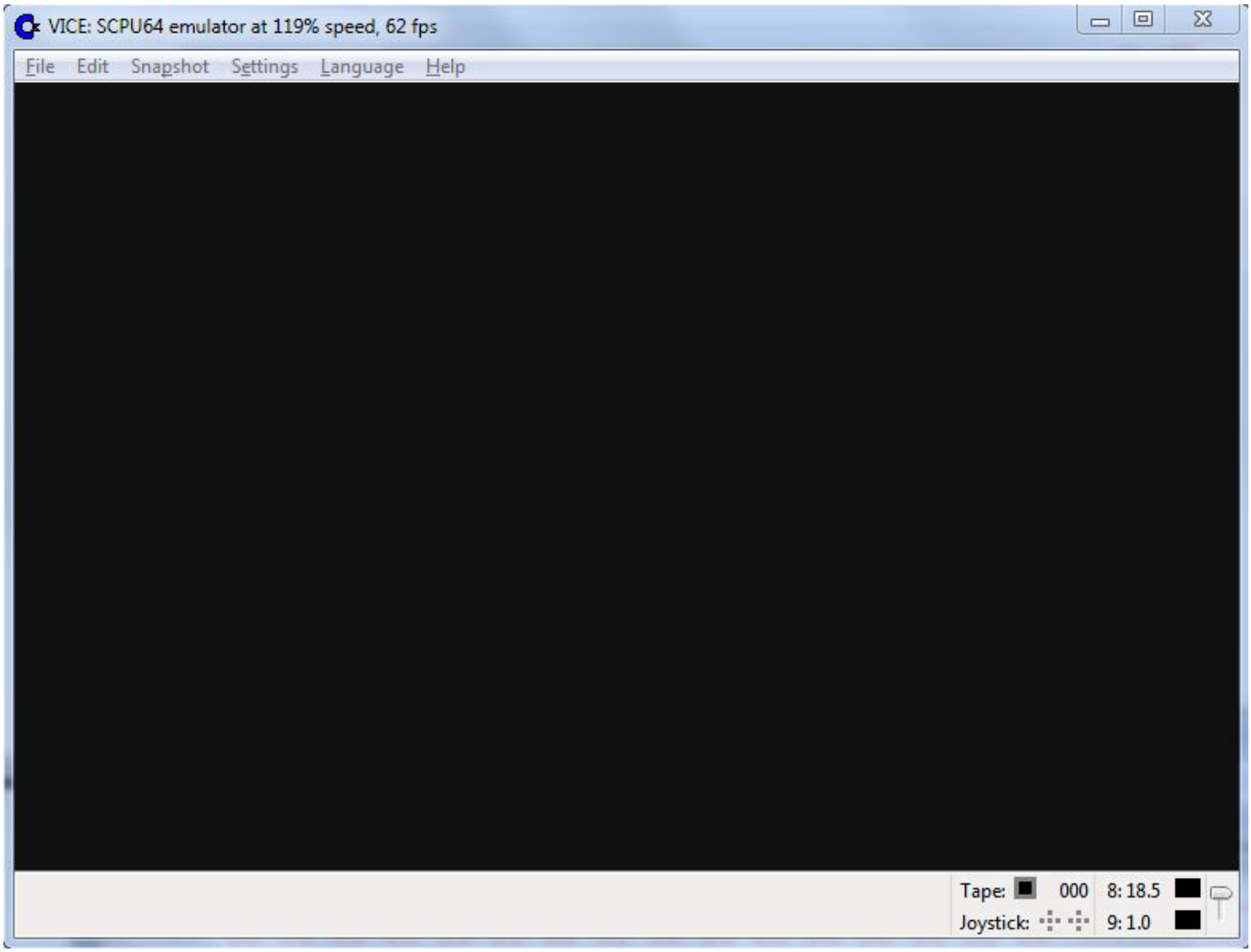

4. Insert payload binary at $1000.



5. Click **Settings/ROM Settings** and load the image into VICE.



6. Click **File/Reset/Hard** and perform a Hard Reset (Ctrl+Alt+R). You should see something like this.




A register display should look like this. Use Alt+M to enter the monitor, then the <R> command.

```
  PB ADDR CREG X    Y    STCK DPRE DB NVMXDIZC E LIN CYC.SB
.;00 1000 ffff 0000 0000 01ff 0000 00 00000111 0 004 052.13
(C:$1000)
```

That's it! We are finished.

SUPERCPU