BIOS Programming in VirtualBox Part 2

So, in the [last document about BIOS programming](https://retroscience.net/BIOS-programming-in-virtualbox.html) we looked at how to get setup to writing a boot loader that will run from a floppy drive on VirtualBox using BIOS. In this document we will look at how we can jailbreak out of the boot sector and make it, so we are not stuck with 512 bytes memory for our code. The way we will do this is by loading up some more data from the floppy drive using BIOS interrupts and then jumping off into our newly loaded code. This will give us much more space for our programs. We will also look at how to use VirtualBox debug tools as they will really come in handy for debugging and testing.

Table of Contents

[BIOS Programming in VirtualBox Part 2 1](#_Toc105930814)

[Prerequisite 1](#_Toc105930815)

[Loading data into memory 1](#_Toc105930816)

[Debugging in VirtualBox 1](#_Toc105930817)

[Launching debug VirtualBox 1](#_Toc105930818)

[Printing out register values 1](#_Toc105930819)

[Printing out memory values 1](#_Toc105930820)

Prerequisite

In order to follow along with this document, you probably will want to check out the [previous document](https://retroscience.net/BIOS-programming-in-virtualbox.html) to get the starter code and your VM setup in VirtualBox.

Loading data into memory

Now that we have a working program, it is time for us to jail-break out of our 512 bytes of memory by jumping to another part of memory which will have code we can execute. There are two things we need to do for do this; (1) write a bit of code that we can jump to at a specific address, and (2) jump to the new address from the boot loader.

We will begin by writing the code we wish to inject into the floppy drive. We could create its own assembly file and in turn its own object file which we can then insert into our floppy image; but for simplicity we will stick to a single assembly file for now.

BITS 16         *; Instruct the system this is 16-bit code*

*; This is the entry point, nothing should happen before this*

*; other than setting the instruction size*

*main:*

        mov ax, 07C0h *; Setup 4KB stack space after this bootloader*

        add ax, 288 *; (4096+515) / 16 bytes (aligned) per paragraph*

        cli     *; Disable interrupts (solves old DOS bug)*

        mov ss, ax *; Assign current stack segment*

        mov sp, 4096 *; Setup our stack pointer*

        sti     *; Enable interrupts (solvs old DOS bug)*

        mov ax, 07C0h *; 07C0h is where our program is located*

        mov ds, ax *; Set data segment to the load point of our program*

        call *run        ; Start the main loop*

*;-------------------------------------------------------------------------------*

*; Constants*

*;-------------------------------------------------------------------------------*

*s\_hi* db "Hello, World!", 0Dh, 0Ah, "- Brent", 0Dh, 0Ah, 00h

*;-------------------------------------------------------------------------------*

*; The main loop of our program*

*;-------------------------------------------------------------------------------*

*run:*

        mov si, *s\_hi    ; Set our si register to point to the hello message*

        call *print      ; Call our print subroutine to print the message*

*; (NEW 1)*

        mov ah, 02h *; Read sectors*

        mov al, 01h *; We want to read 1 sector*

        mov ch, 00h *; From cylinder number 0*

        mov cl, 02h *; Sector # (Boot 1 [index starts at 1 not 0])*

        mov dh, 00h *; Head number 0*

        xor bx, bx

        mov es, bx *; es should be 0*

        mov bx, 7E00h *; To address; program boot address + 512*

        int 13h *; Read floppy drive interrupt*

*; (NEW 2)*

        push 7E00h *; Don't let the assembler treat this as local offset*

        ret     *; Jump to the pushed address*

*;-------------------------------------------------------------------------------*

*; Print string subroutine (null terminated string print)*

*;-------------------------------------------------------------------------------*

*print:*

        push ax *; Save the current value of the AX register*

        mov ah, 0Eh *; Our first BIOS interrupt:  Teletype output*

        mov bl, 0Fh *; Color*

.repeat:

        lodsb   *; Load next character into AL register*

        cmp al, 00h *; Check if we are at end of string (0 = end of string)*

        je .*done        ; If AL is 0 then jump to done label*

        int 10h *; BIOS interrupt 10h (0x10 or 16 in decimal)*

        jmp .repeat *; Continue to next character in the string*

.*done:*

        pop ax   *; Restore the value to the AX register*

        ret     *; Return to caller location*

*;-------------------------------------------------------------------------------*

*; Boot loaders are 512 bytes in size so pad the remaining bytes with 0*

*;-------------------------------------------------------------------------------*

times 510-($-$$) db 0 *; Pad (510 - current position) bytes of 0*

dw 0xAA55       *; Boot sector code trailer*

*; (NEW 3)*

*;-------------------------------------------------------------------------------*

*; Code which is loaded to 7E00h (512 bytes after 7C00h- our program boot addr)*

*;-------------------------------------------------------------------------------*

*wild\_zone:*

        mov si, *freedom* *; Set our si to point to the hello message*

        call *print*       *; Call our print subroutine to print the message*

.loop:

        jmp .loop       *; Infinite loop to hold control of the computer*

*;-------------------------------------------------------------------------------*

*; Constants*

*;-------------------------------------------------------------------------------*

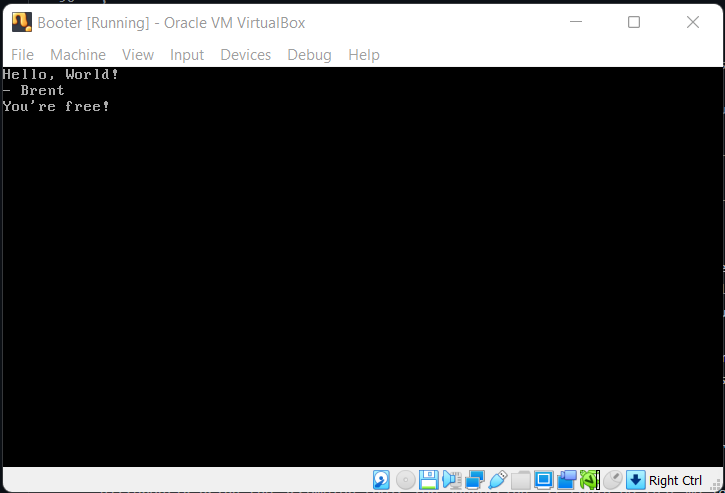
*freedom* db "You're free!", 00h

There are only 3 new sections of code in this from the previous document’s examples. The code below [(NEW 1)](#line40) is the code that we need to load up more data from the floppy drive into memory. The important interrupt here to note is the int 13h interrupt. This will take the arguments provided and load up data as specified. Technically this operation could fail, so you would want to run it a few times upon failure to be sure (resetting the disk with AH=00h between attempts).

At this point we have loaded some more code into memory and are ready to jump to it; this brings us to [(New 2)](#line52new2). Here we do not simply do jmp 7E00h because the assembler will assume that is an offset from our current position in code, so we will do a little trick by pushing the address onto the stack, and then calling ret which will pop an address from the stack and jump to it.

Lastly, we have [(New 3)](#line92new3) which is a simple piece of code that will just print the string “You’re free!” to the screen. **WARNING!** We are using call print here; the assembler may choose to use the relative offset to call the routine. If you are working in another file, or even this file in the future, I’d handle that a little differently using the assembler tools for addressing. It could be that we load the new sectors from the floppy to some memory that is not directly consecutive with the boot code, so then the relative address would be incorrect.

At this point you should see the new print when you build and run!

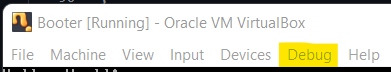


Debugging in VirtualBox

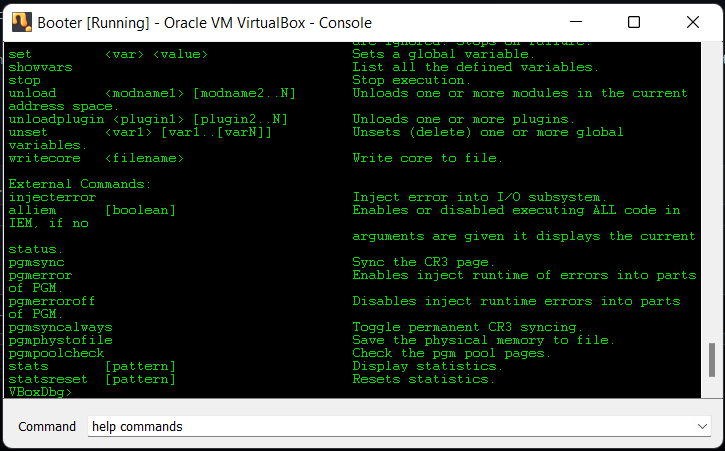
Sometimes it is **exceptionally** difficult to debug these kinds of programs, but luckily, we are using VirtualBox! It provides many tools for us to be able to debug our programs and see the state of things. Now I am no master at debugging in VirtualBox, but I can at least show how to get the tools up and what a couple of nice commands are.

Launching debug VirtualBox

If you have VirtualBox in your path, you can call VirtualBoxVM --startvm "Booter" --dbg. You will want to replace “Booter” with whatever your virtual machine name is. This will add an extra “Debug” menu option in the menu bar of the running instance, and conveniently launch your VM.

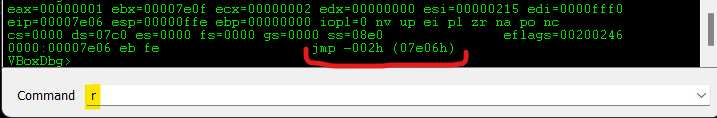


Your best friend here will be the option Command Line... so select this option. You will be presented with a window below your VM to enter commands and view responses. The most important command to remember is help commands, which will list out all the commands you can run.



Printing out register values

There are two helpful commands, and the first one is r. If you submit the r command, you will be presented with all the current values of the registers as well as a helpful print of the current instruction that the system is currently executing.



Printing out memory values

The second command that is super helpful is the ability to review what you currently have in memory. If you type db [addr] and replace [addr] with a memory address, it will print out all the bytes for this address and the following addresses. So, for example, we can print out db 7C00 and 7E00 to print out our boot loader program and loaded sector respectively.

