

# BOS Programmers Guide

## Background

BOS is a very lean and bare bones, so much so, the language used to program the OS is x86\_64 assembler. Although some may feel that assembler is complex, intimidating, and takes too long. I have found in my experience in creating BOS that using assembler, after an initial learning curve, it is really not that hard to grasp. The main advantage of using assembler is that all the power and efficiencies of assembler are at your disposal.

You have the ability to create anything you want, without the need for anyone else's libraries or learning someone else's syntax. You have complete control and you can create whatever you want (reference to Flynn for those that understand the reference).

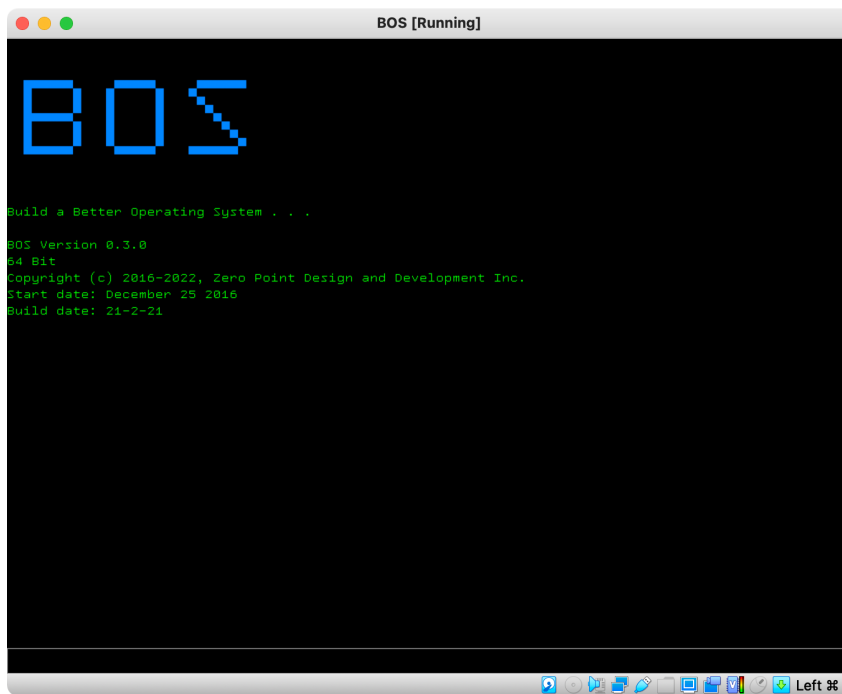


Figure 1 - BOS

## Setup

BOS is written in x86 assembler using Netwide Assembler (<https://www.nasm.us/>), using a text editor that highlights assembler syntax, and a popular Linux distribution based on Ubuntu.

BOS supports creating user programs in either assembler or C. The advantage of using assembling is that you will have more control of your program. While the advantage of creating user programs in C is that you will not need to learn assembler and can stick to what you know.

For you to write code for BOS I would suggest the following (however, you can choose whatever tools you wish);

1. A Linux distribution - Ubuntu. I use a Ubuntu based distro.
2. A text editor - Atom.
3. For those that will code in native assembler - nasm (sudo apt install nasm).
4. For those that will code in 'C' – gcc and binutil.
5. And the VM VirtualBox Expansion Pack. This is needed when you want to enable debugging in VirtualBox.

## Graphics

All user programs can access graphics memory at virtual address 0x40000000. This is by design to allow the developer to write directly to the graphics card.

## Programming in Assembler

The following section describes what you will need to do in order to create user programs in assembler.

### Assembling Code

Anyone that has coded in assembler knows that code is not compiled, it is assembled (hence the name) and converted to machine code (numbers the CPU understands). The basic command to assemble your code is the following:

```
nasm <source_file> -f bin -l <source_file.lst> -o <output_file>
```

Example:

```
nasm hello_world.asm -f bin -l hello_world.lst -o hello_world.app
```

Where:

- <source\_file> is the text file to be assembled.
- -f bin is the file format. BOS uses a simple flat binary file (keep it simple).
- -l <source\_file.lst> generates a list file that will show the memory addressing, machine code and source code. This is needed to debug your code.
- -o <output\_file> is the binary file itself. You can view this file using a hex editor (e.g. hd hello\_world.app).

To make assembling more efficient, I would recommend creating a script file to assemble your code and naming the script file something very easy (e.g. ab). Below is a sample script you could use:

```
#!/bin/bash
# Script file to assemble the hello world program.
NAME="hello_world"
nasm $NAME.asm -f bin -l $NAME.lst -o $NAME.app
```

NOTE: Make sure the script file has execution permissions:

```
chmod +x ab.sh
```

## Source Code

BOS uses flat binary files to run applications; this file type was chosen because it left full flexibility to the developer. That is, format the file however you like and use it. For example, looking for certain values as inputs, looking for values at specific memory offsets (much like headers in an ELF file), etc.

There are a few rules that need to be followed in the source code:

1. All source code must begin with:

```
org 0x70000000
bits 64
jmp ENTRY
nop
nop
```
2. User programs must begin at address 0x70000000, otherwise the program will generate a memory protection fault.
3. BOS is a 64 bit operating system, therefore the source file must use the 'bits 64' directive.
4. Jump to the first executable instruction. This is actually more of a recommendation than it is a rule. Using a jump statement (jmp) allows the developer to have any data they wish at the top of the file. If you don't have a jump statement, whatever is after 'bits 64' will be treated as executable machine code. This can lead to some very interesting outcomes which may or may not have the desired effect.
5. The 'nop' assembler statements are not required, I use them to fill CPU prefetching.
6. A program must exit using the `Exit` routine in `lib_app.asm` or calling the `exit` system call. If this is not done, the program will generate a Memory Protection Fault.

System call to exit a program:

```
mov rdx,0x0
int 0xFF
```

## Programming in C

Because BOS is a new operating system, a most often used approach is to create a cross compiler that will compile code on a host system (e.g. Linux desktop) for a target system (in this case BOS). However, there is an easier approach and all that is really needed is to have a custom Makefile and link script. Although optional, it is highly recommended to also use the BOS library file that will front all system calls. This file uses embedded inline assembler to make the system calls and return the results. For those with advanced programming skills, you are encouraged to view and modify this file (and feel free to share the results back with the community).

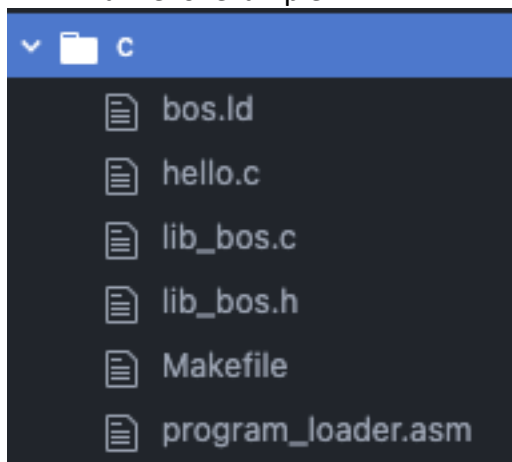
## Requirements

It is recommended to develop user programs using a Linux (virtual) desktop that has 'gcc' and 'binutils' installed. The following files are required (and provided):

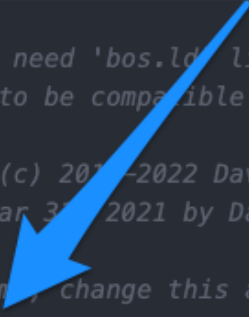
1. **Program\_loader.asm**: This is an assembler program used to load the actual user program (equivalent to crt0 for those that are aware of this file). This file tells the OS what function to call when the program starts. In C, this program is 'main'.
2. **BOS.ld**: This is a linker script used to link everything together. The main purpose of this script is to tell the compiler to set the origin address to 0x70000000 and set the output type to a flat binary file.
3. **Lib\_bos.c / Lib\_bos.h (Recommended)**: This is the front end to syscalls and whatever else is needed.
4. **Makefile**: This file is used to 'make' and 'make clean' the user program. It will compile and link everything together. This file assumes you will be using lib\_bos and therefore compiles it into your user program. If you intend not to use lib\_bos then update this file accordingly.

## Compiling

1. When you have completed your source code and you are ready to compile, make sure you have all the files needed in the same directory as the source file. The image below is a 'hello' example.



2. Modify the Makefile and change the NAME variable to the name of your source code file

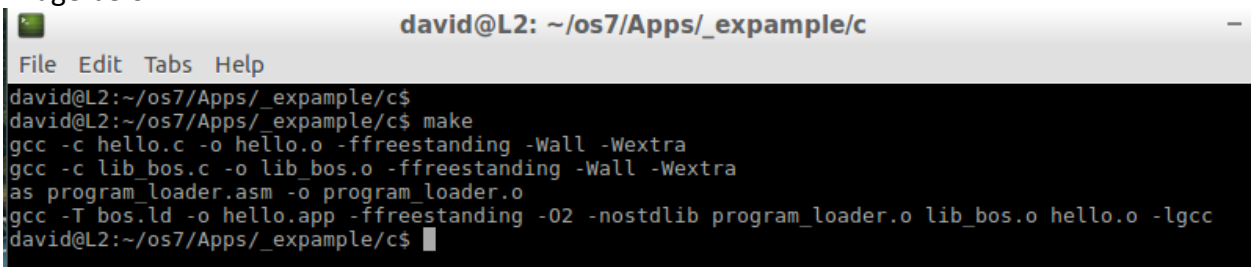


```
# Makefile for a BOS application that will use standard 'gcc' and 'as' on Linux.
# Note:
#   You will need 'bos.ld' link file and program_loader.asm to build the
#   program to be compatible with BOS.
#
# Copyright (c) 2017-2022 David Borsato
# Created: Mar 30, 2021 by David Borsato

# Program name, change this as needed.
NAME = hello

# Defining variables for GCC and AS is to prepare for a future BOS specific
# cross compiler.
GCC = gcc
ASM = as
```

3. Launch a terminal and navigate to the directory where the source and files are, run the command 'make' to compile. If there are no errors, you will see output similar to the image below.



```
david@L2: ~/os7/Apps/_example/c
File Edit Tabs Help
david@L2:~/os7/Apps/_example/c$
david@L2:~/os7/Apps/_example/c$ make
gcc -c hello.c -o hello.o -ffreestanding -Wall -Wextra
gcc -c lib_bos.c -o lib_bos.o -ffreestanding -Wall -Wextra
as program_loader.asm -o program_loader.o
gcc -T bos.ld -o hello.app -ffreestanding -O2 -nostdlib program_loader.o lib_bos.o hello.o -lgcc
david@L2:~/os7/Apps/_example/c$
```

## Copy Binary File to BOS

Once a binary file is created, it needs to be copied to BOS' App directory (all user programs are placed in this directory). Having all user programs in one location is done so by design. This makes it very easy to determine which files you are aware of and which files you are not aware of.

Copying binary files to BOS is done by running a program on BOS that will allow you to use a browser to drag and drop your files into BOS:

1. Start the BOS VM.
2. At the BOS command box type 'run load\_app.bin'. This launches a program that will allow you to connect with a browser. You will see something similar to the image below.

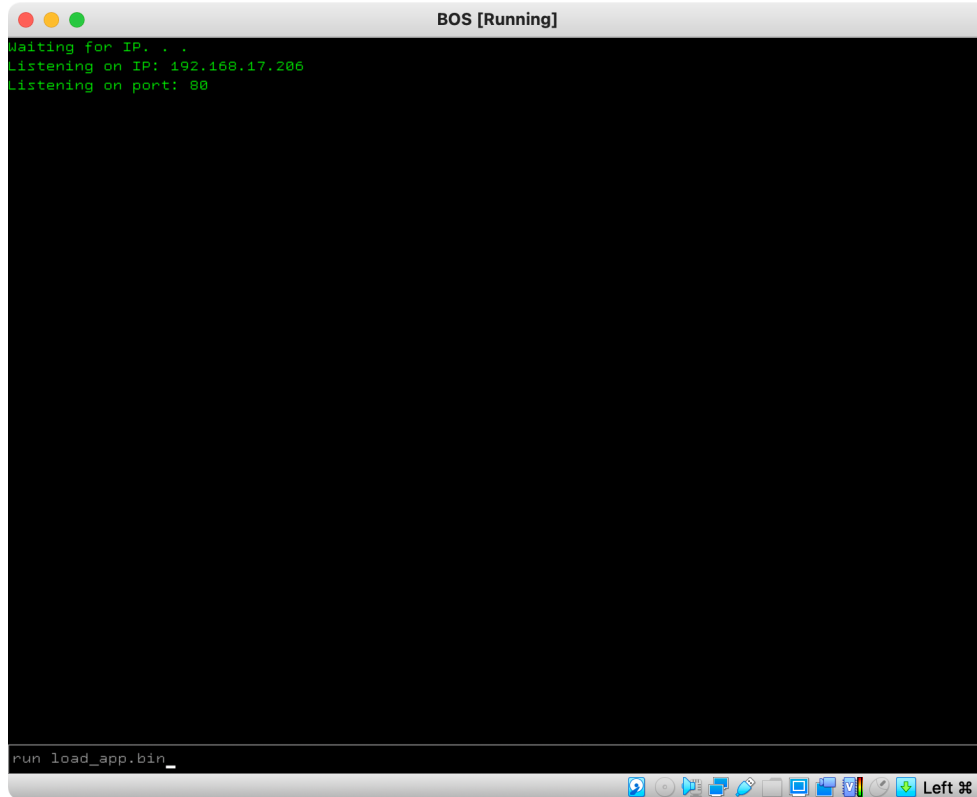


Figure 2

3. Referring to the 'Listening on IP' line in figure 2, make note of the IP address. This is the IP address you will put in the browser. If you happen to change the screen (e.g. clear), you can also get the IP address by typing the command 'ip show' and referring to the 'IP Address' line.

4. Launch a browser and type the IP address in the address bar. You should see something similar to figure 3.

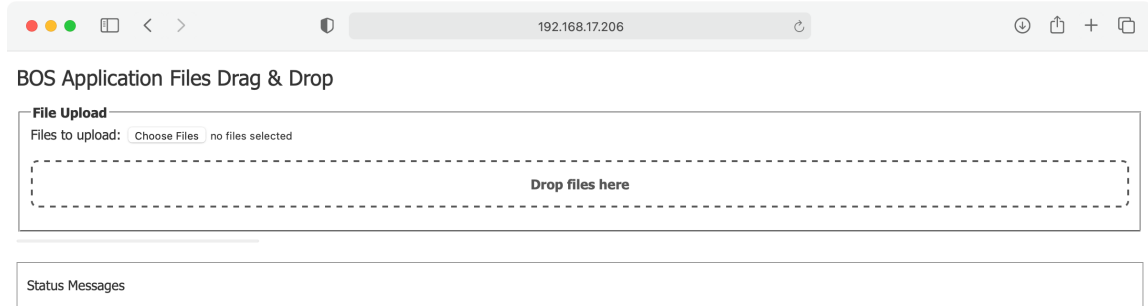


Figure 3

5. From a file window, drag the binary file to the 'Drag files here' box.

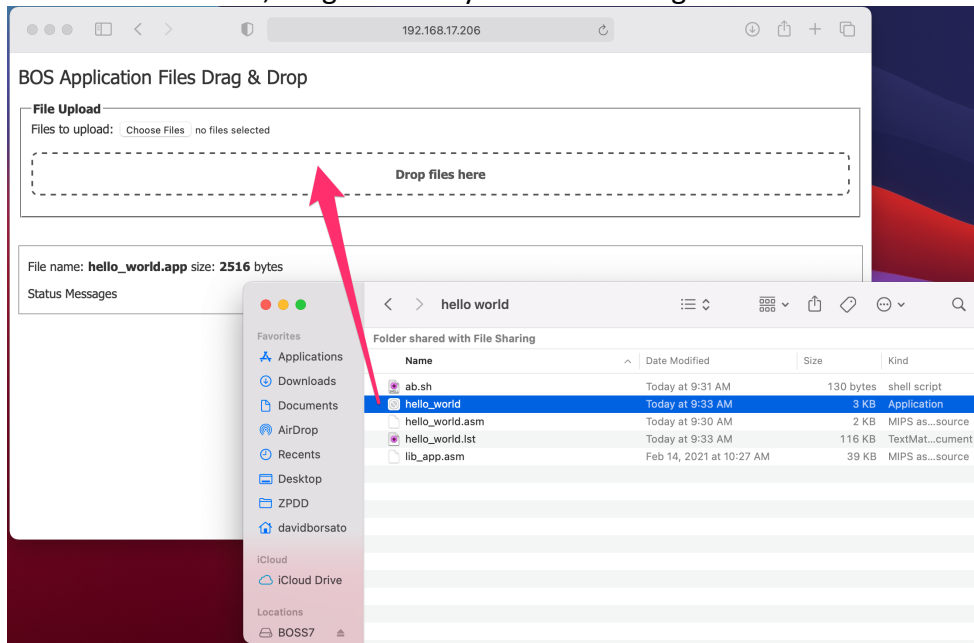
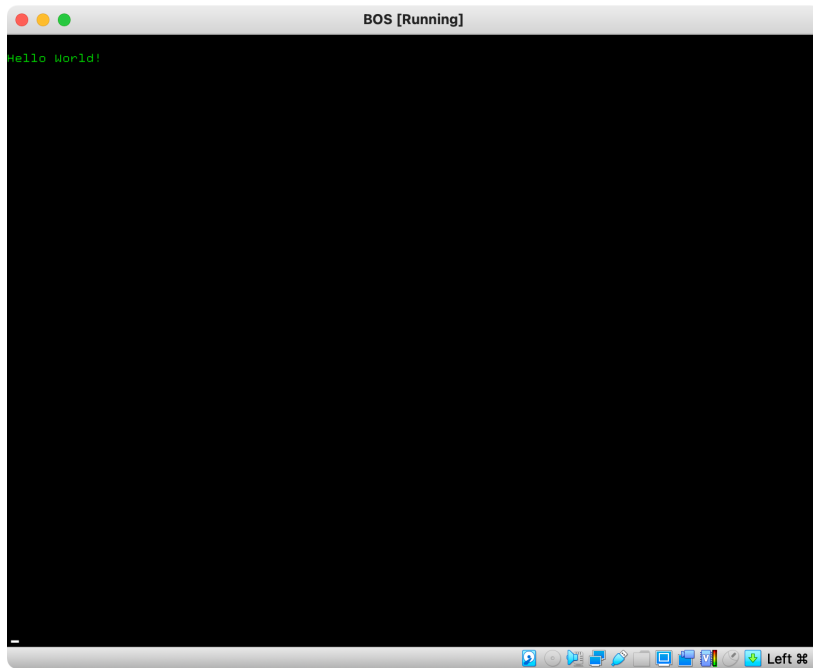


Figure 4

6. Go back to BOS.
7. To run the program type 'run <program\_name>'. In this case the command would be 'run hello\_world.app'.





8. Done.

## Debugging Code

All developers need to be able to debug their code. One common way is to print information to the screen, which is still a good approach with BOS. The other way is to stop the virtual machine and inspect code, registers and memory directly using VirtualBox's debug mode. This mode will make use of the list file.

Looking at the list file for the 'hello\_no\_lib' program, it has the following parts:

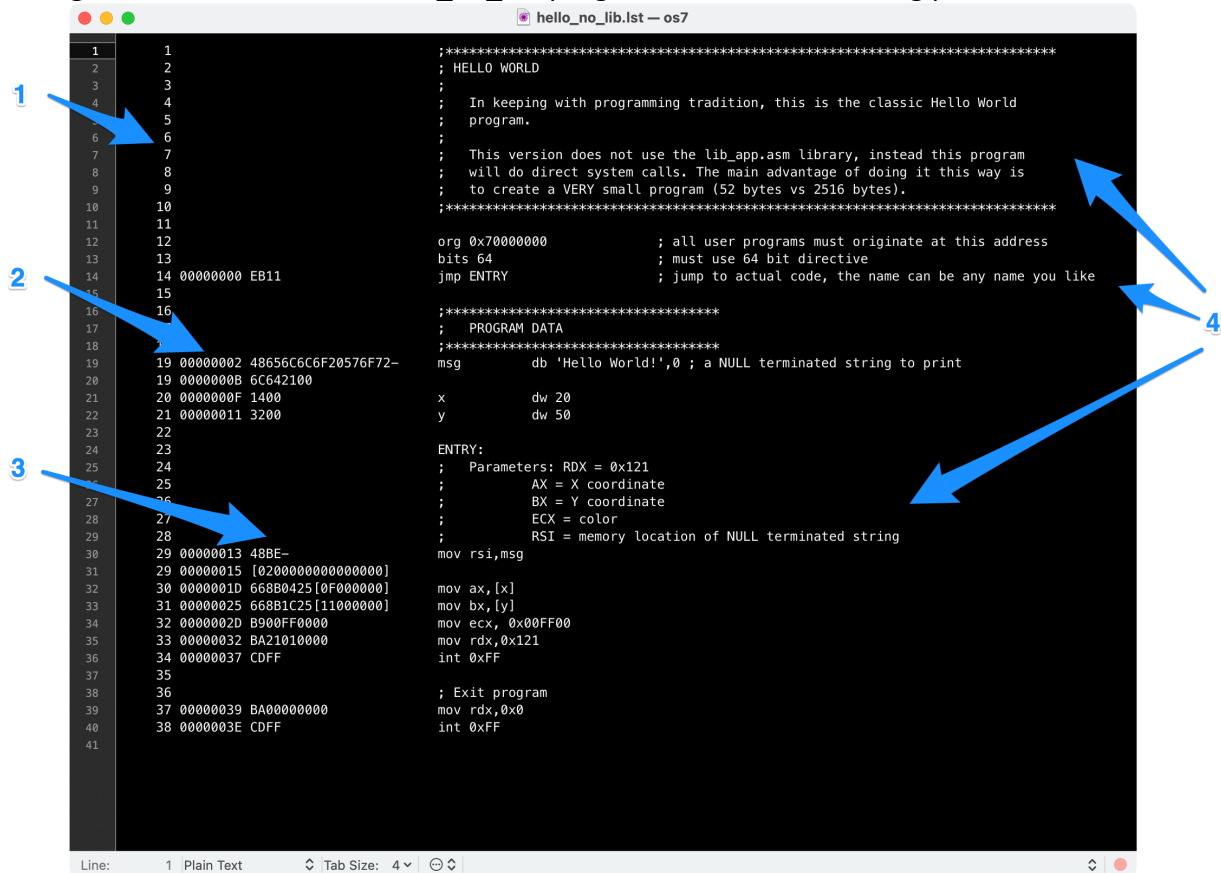


Figure 5

1. Line numbers: Each line is numbered, this is not used.
2. Memory offset: This is important and will be discussed in more detail.
3. Machine code: Assembled code into actual machine code, not really used but still informative.
4. Source code: Source code itself, this is used to determine break points and helps when stepping through the code.

Using the information in the list file, we can use VirtualBox's debug mode to set a break point that will stop the OS. When the OS is stopped, you can now inspect the register's values and memory to determine if the program is doing what you expect.

The way a break point is set is by specifying a memory address to stop at. This is why the memory offset in the list file is important. Using the listing in figure 5, if we wanted to stop at the line:

```
mov rsi,msg
```

The memory address to stop at will be 0x70000013. This is determined by doing the following:

1. Using the source code (item 4), find the line you want to line at.
2. Get the memory offset value (item 2) and add it to 0x70000000. This is the virtual address for all user programs and cannot be changed.
3. Therefore,  $0x70000000 + 0x13 = 0x70000013$ . Note this is a hex number.

If you wanted to stop at the line 'mov rdx,0x121', the memory address to stop at would be 0x70000032. And so on.

Starting a VM in debug mode is different than normally starting a VM. Normally you would start a VM from the VirtualBox main screen by highlighting the VM and selecting 'Start', double clicking the VM, etc.

To start a VM with debugging enabled, this must be done at the command line. And, I have found this works best when you have VirtualBox running while you use the command line.

The command to start a VM in debug mode is:

```
VirtualBoxVM --debug-command-line --start-dbg --startvm <VM_NAME> &
```

For BOS use:

```
VirtualBoxVM --debug-command-line --start-dbg --startvm BOS &
```

The BOS VM will launch in a paused state. This gives you the ability to set break points before the OS runs. See figure 6.

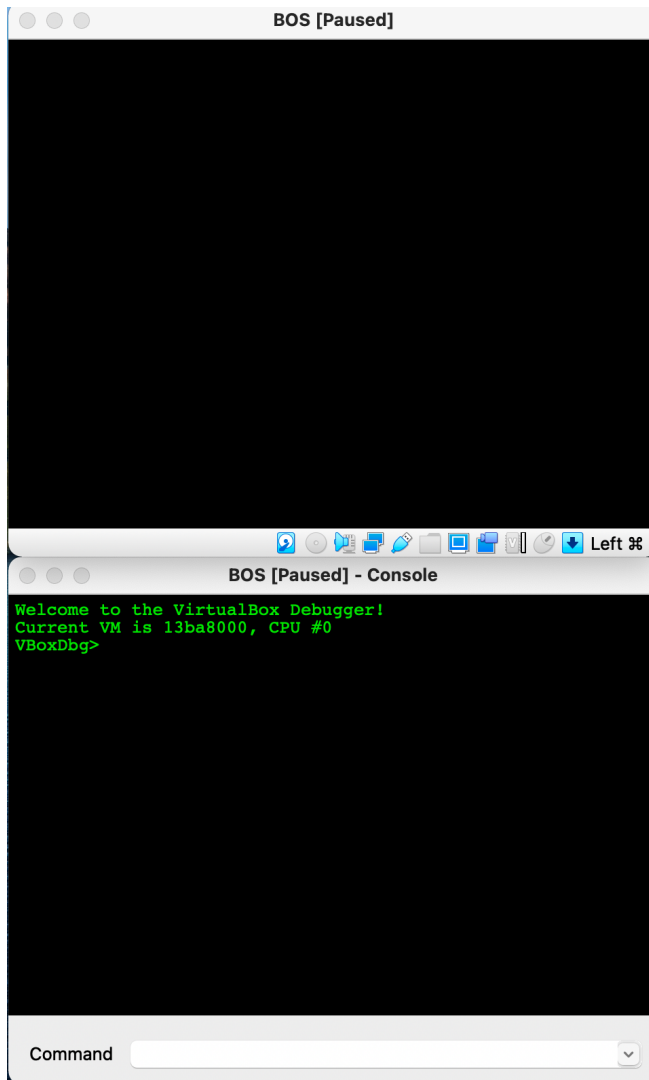


Figure 6

In the VirtualBox Console Command box, type this command to set a break point:  
ba x 1 0x70000013

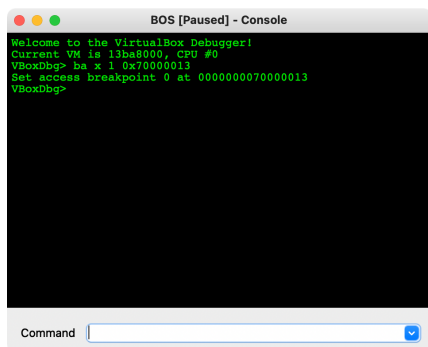


Figure 7

With the break point set, unpause the VM by selecting from the top menu Machine->Pause. Or Host+P on a MAC. BOS will startup as normal, the next step is to run the program:  
`run hello_no_lib.app`

The program will now stop at the break point set earlier. Refer to the Console window in the figure below

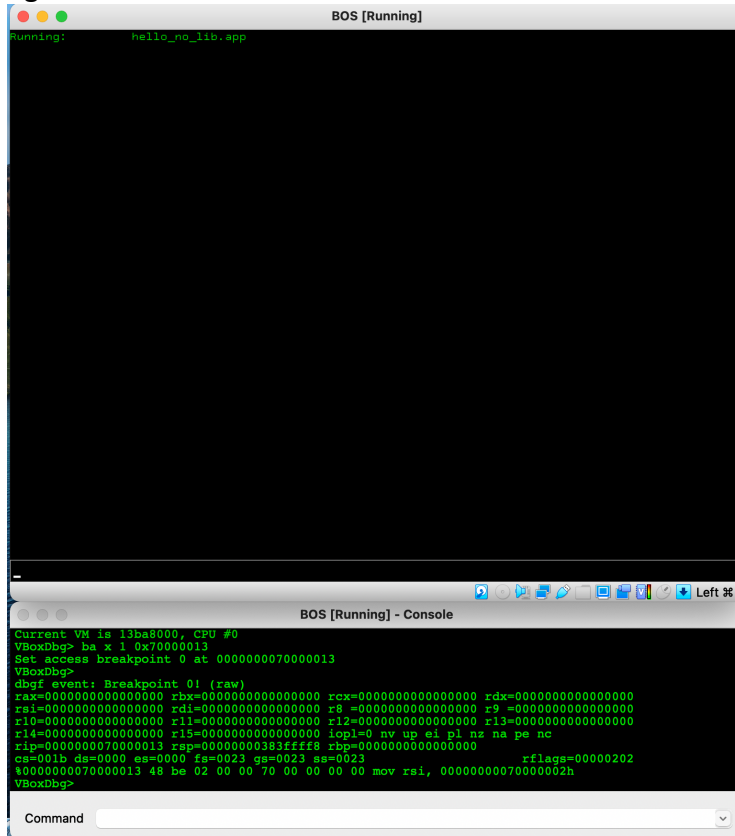


Figure 8

Figure 8 shows that in the Console window, the code has stopped and displays the value of the registers. At this point you can now issue commands to display memory, step through the code, etc.

Below are a common commands used:

```
ba x 1 <addr>      sets a break point at the address
bc <# | all>       deletes a breakpoint
bd <#>             disables a breakpoint
be <#>             enables a breakpoint
bl                 list break points
```

```
g                 continue execution
```

```
p                 step over
```

```
t                 single step
```

```
t                 [count] [cmds]
```

```
tr
```

```
tracing & stepping (no code executed).
```

```
Trace .
```

```
Toggle displaying registers for
```

ta	<addr> [count] [cmds]	Trace to the given address.
tc	[count] [cmds]	Trace to the next call
instruction.		
tt	[count] [cmds]	Trace to the next return
instruction.		
p	[count] [cmds]	Step over.
pr		Toggle displaying registers for
tracing & stepping (no code executed).		
pa	<addr> [count] [cmds]	Step to the given address.
pc	[count] [cmds]	Step to the next call instruction.
pt	[count] [cmds]	Step to the next return
instruction.		

NOTE: use list file to find where to disassemble from

d	dump memory
dd	dump memory, double words (easier to read)
u 1028d7	unassembles code at 0x1028d7
u64	unassemble 64 bit code
r	show registers