

Realmode Assembly: Writing Bootable Stuff

September 24, 2018

1 Part 1: Theory and Concepts

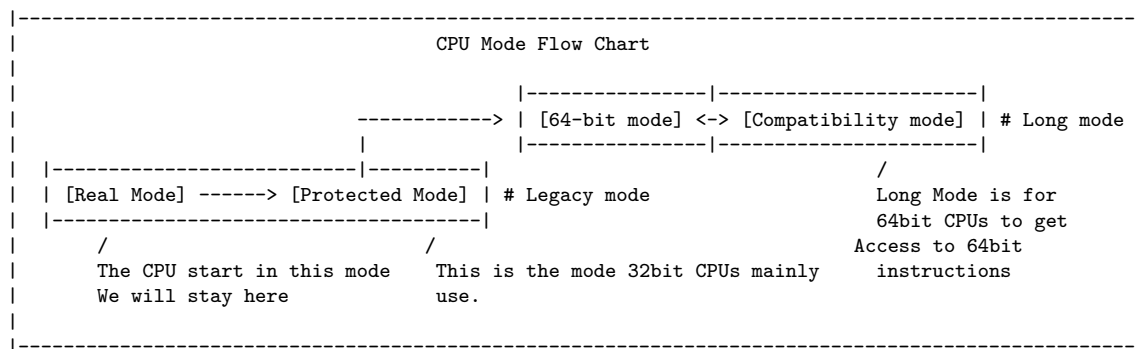
1.1 What is Realmode?

Real mode is the operation mode all x86 compatible CPUs start in. Originally CPUs were only able to use real mode, later CPUs switched to protected mode (which is the main operation modes for modern Intel processors and allows for example Virtual Memory allocation and instruction set restriction through privilege levels) but starting in real mode was kept in for legacy reasons.

It doesn't supply fancy stuff like virtual addresses so all addresses used are actual physical addresses also we won't be able to access 64-bit instructions as they are only accessible in Long mode (the mode x86-64 mainly operate in).

It provides unlimited direct access to memory, I/O and peripheral hardware which means we have control over almost everything. We will only have 1MiB of accessible memory (actually less) but that will be enough as we won't write too long code. The BIOS provides lots of handy interrupts to interact with I/O and peripherals.

It also provides drivers for devices so we don't need to worry about our mouse or keyboard not working (except you have some weird hardware). Also important to know is that the default CPU operation length is 16bit so we rarely going to use 32bit registers or operations.



1.2 BIOS

The BIOS(Basic Input/Output System) is the system that's responsible for initializing hardware on startup of the computer and responsible for providing generalized services for kernel and programs to easily interact with hardware (mouse, keyboard, display). It comes pre-installed on the motherboard and is the first software run after power-on. All interrupts we will use will be directed to the BIOS so it will be a core component of our real mode assembly.

1.3 Bootloader/MBR

The BIOS checks bootable devices for the boot signature in their first sector and writes this segment, if the boot signature was found, to the physical address 0x0000:0x7c00 (0st segment at address 0x7c00). This is the Master Boot Record, it's limited to 512 bytes and it's supposed to pick the hard drive and partition to boot from, write the bootloader of that partition into memory and jump to it so it can initialize the kernel loading.

That won't be necessary for our small Operation Systems as we won't have partitions or a proper file system or anything in that matter so we will just use the MBR to load in the kernel into RAM and to give control to it.

1.4 Kernel

The kernel is the main part of an operation system. In the future code of this walk-through it will be responsible for everything, it will be the main place for our code. (I'm actually not sure if that technically counts as a kernel)

1.5 So what's the plan?

First we will write a Master Boot Record, that copies the kernel from the hard drive to memory. We will do that using BIOS Interrupt 0x13 which is responsible for reading and writing operations which involve drives.

After that we will load into our kernel and send our monitor a nice "Hello World" message using BIOS Interrupt 0x10 which will be our method of writing text and drawing pixels to the screen.

```
|-----|
|               1 MB Memory we have available               |
|-----|-----|-----|-----|
| MBR      | Kernel | Free Space for us to use| BIOS and Hardware Area |
| 0x7c00   | 0x8000 | ~640KB for us to use   | ~360KB                 |
|-----|-----|-----|-----|
```

```
[BIOS] --Loads--> [Master Boot Record / Bootloader] --Loads--> [Kernel] --Prints--> [Display]
```

(If you are interested in understanding more modern boot sequences (after floppy disks) you can check the [last source&reference link](#))

2 Part 2: Hello World Bootloader

2.1 Writing a loader for the kernel:

So to start we are going to write a minimalistic bootloader that just loads the kernel and then gives control to it. We will improve the bootloader later and make it more abstract and reliable but for the explanation this will be enough.

Let's go: We will need to execute two interrupts, one to reset the disk system which will ensure we will read the right part and one to read the kernel from hard drive to RAM. As already explained in the last article we will use the interrupt 0x13 which decides which disk operation to execute based on the ah register (higher byte of ax, which is the lower word of eax).

For the disk resetting we will use the first function of the interrupt 0x13, so ah will be set to 0. I will use the syntax `int 0x13,0` to refer to this interrupt to make my life a bit easier. `int 0x13,0` takes two parameters ah = 0 and dl = drive number (a number representing a specific drive. for example: 0x80 = drive 0). Luckily the BIOS (or better said almost every BIOS) puts the drive number of the device currently booting from (in our case the hard drive the kernel is on or an emulated device) into dl before giving control to the MBR, so we will assume that the drive number is already in dl which will be the case most of the time.

int	ah	dl	Description
int 0x13	0	drive number	Resetting Disk System

As already explained previously the bootloader will be loaded into 0st segment at address 0x7c00 so we also have to tell NASM to calculate offsets starting from that address. Also let's tell NASM that we are in 16bit mode so it tells us when we use illegal instructions for this mode and knows how to correctly refer to memory.

```
org 0x7C00 ;NASM now calculated everything with the right offset
bits 16
mov ah, 0
int 0x13 ; int 0x13,0 dl = drive number
..
```

Now that the disk system is reset we can start reading from the correct offset of the hard drive we boot from. For that we will use `int 0x13,2`. It takes an address within the RAM to write in as argument into bx, we will just write it to 0x8000 as it's free space. The al register will contain the amount of sectors to read, this will be the size of our kernel and as our HelloWorld Kernel will be small we just write a 1 into al (1 sector = 512 bytes, size of kernel has to be multiple of 512 bytes). If we increase the size of our kernel we have to increase this number as well and because it's pretty annoying to change this every time we make the kernel bigger so we will later look at ways to make this easier for us. The address to read from is in the Cylinder-Head-Sector (CHS) format (which is a pretty old format for addresses on hard drives) so we have to do some converting (also knowing the math behind those numbers helps reading from other segments):

(Following values are true for 1.44 MB 3.5" Floppy Disks , the theory applies for every hard drive. Obviously this memory layout doesn't make sense for USB-Devices and similar but as it's the way the BIOS works we will later add a Standard BIOS Parameter Block which will define the following values for the BIOS)

Each Sector has a size of 512 (BytesPerSector) bytes. Sectors start with 1. Each Head contains 18 (SectorsPerTrack) sectors. Each Cylinder (also called Track) contains 2(Sides) Heads. There are 2(Sides)

Cylinders.

Using this we can calculate the CHS-Values for the logical segment(continuous numeration starting from 0) we want to read from. The bootloader is at logical segment 0 (as it's the first segment on the hard drive). After that the kernel follows at logical segment 1 (as it's directly after the bootloader on the hard drive and the bootloader is exactly 512 bytes (1 segment) in size.)

int	ah	dl	ch	dh	cl	Description
int 0x13	2	drive number	Reading Track	Reading Head	Reading Sector	Read from Hard drive

Logical segment `ls = 1`

=>

Cylinder is saved in `ch`

`ch = Cylinder/Track = (ls/SectorsPerTrack)/Sides = 0`

Head is saved in `dh`

`dh = Head =(ls/SectorsPerTrack)%Sides = 0`

Sector is saved in `cl`

`cl = Sector = (ls%SectorsPerTrack)+1 = 2`

Our new code now looks like this:

```
..
mov bx, 0x8000      ; bx = address to write the kernel to
mov al, 1           ; al = amount of sectors to read
mov ch, 0           ; cylinder/track = 0
mov dh, 0           ; head = 0
mov cl, 2           ; sector = 2
mov ah, 2           ; ah = 2: read from drive
int 0x13            ; => ah = status, al = amount read
```

Ok next let's add the remaining parts to make this bootloader work:

```
..
; pass execution to kernel
jmp 0x8000
;$ = address of current position
$$ = address for start of segment
;($-$$) = amount of already filled bytes of this segment
;pads everything from here up to 510 with 0's
;also gives compiler errors if not possible which
;might happen if we already wrote more than 510 bytes in this segment and
;thus causes ($-$$) to be negative
;this is very useful as it makes sure that the resulting binary
;has a size multiple of 512 which is required to make everything work
times 510-($-$$) db 0
;Begin MBR Signature
db 0x55 ;byte 511 = 0x55
db 0xAA ;byte 512 = 0xAA
```

Ok now our bootloader is finally done but as we don't have a kernel yet we can't test it.

Complete MBR Code:

```
org 0x7C00
;Reset disk system
```

```

mov ah, 0
int 0x13 ; 0x13 ah=0 dl = drive number

;Read from harddrive and write to RAM
mov bx, 0x8000 ; bx = address to write the kernel to
mov al, 1 ; al = amount of sectors to read
mov ch, 0 ; cylinder/track = 0
mov dh, 0 ; head = 0
mov cl, 2 ; sector = 2
mov ah, 2 ; ah = 2: read from drive
int 0x13 ; => ah = status, al = amount read
jmp 0x8000
times 510-($-$$) db 0
;Begin MBR Signature
db 0x55 ;byte 511 = 0x55
db 0xAA ;byte 512 = 0xAA

```

2.2 Writing a kernel for the loader:

Now that we have a bootloader that loads our kernel let's start writing our kernel. The kernel is supposed to print out a "Hello World" message and then halt/stop everything.

Printing something to the display means we need a way to interact with it. BIOS Interrupt 0x10 will help us here as it's responsible for all kinds of video services (printing characters, drawing pixels, ..). We will print the string "Hello World" character after character using int 0x10,0xE which takes a single character (ASCII) in register al, the page to write to in bh (there is enough memory to have a few text pages for quick swapping, default is page 0) and color attributes in bl (but this is only displayed if we are in a graphical mode with which we will mess later).

int	ah	al	bh	bl	Description
int 0x10	0xE	ASCII Character	Page to write to	Color Attribute	Print a character to the screen

So our print character functions should look like this:

```

printCharacter:
    ;before calling this function al must be set to the character to print
    mov bh, 0x00 ;page to write to, page 0 is displayed by default
    mov bl, 0x00 ;color attribute, doesn't matter for now
    mov ah, 0x0E
    int 0x10 ; int 0x10, 0x0E = print character in al
    ret

```

Given how to print a single character the remaining code for printing a string is pretty simple:

```

printNullTerminatedString:
    pusha ;save all registers to be able to call this from where every we want
.loop:
    lodsb ;loads byte from si into al and increases si
    test al, al ;test if al is 0 which would mean the string reached it's end
    jz .end
    call printCharacter ;print character in al
    jmp .loop ;print next character
.end:
    popa ;restore registers to original state
    ret

```

Now we just have to put it together by telling NASM the right offset and instruction size and actually calling the `printNullTerminatedString` function. Note that I added a padding again to ensure that the final kernel has a size multiple of 512 as it might result in problems reading from the hard drive if the size isn't correct.

```
org 0x8000
bits 16
mov si, msg
call printNullTerminatedString

jmp $ ; this freezes the system, best for testing
hlt ;this makes a real system halt
ret ;this makes qemu halt, to ensure everything works we add both

printCharacter:
;before calling this function al must be set to the character to print
mov bh, 0x00 ;page to write to, page 0 is displayed by default
mov bl, 0x00 ;color attribute, doesn't matter for now
mov ah, 0x0E
int 0x10 ; int 0x10, 0x0E = print character in al
ret

printNullTerminatedString:
pusha ;save all registers to be able to call this from where every we want
.loop:
    lodsb ;loads byte from si into al and increases si
    test al, al ;test if al is 0 which would mean the string reached it's end
    jz .end
    call printCharacter ;print character in al
    jmp .loop ;print next character
.end:
popa ;restore registers to original state
ret

msg db "Hello World!"
times 512-($-$$) db 0 ;kernel must have size multiple of 512
;so let's pad it to the correct size
```

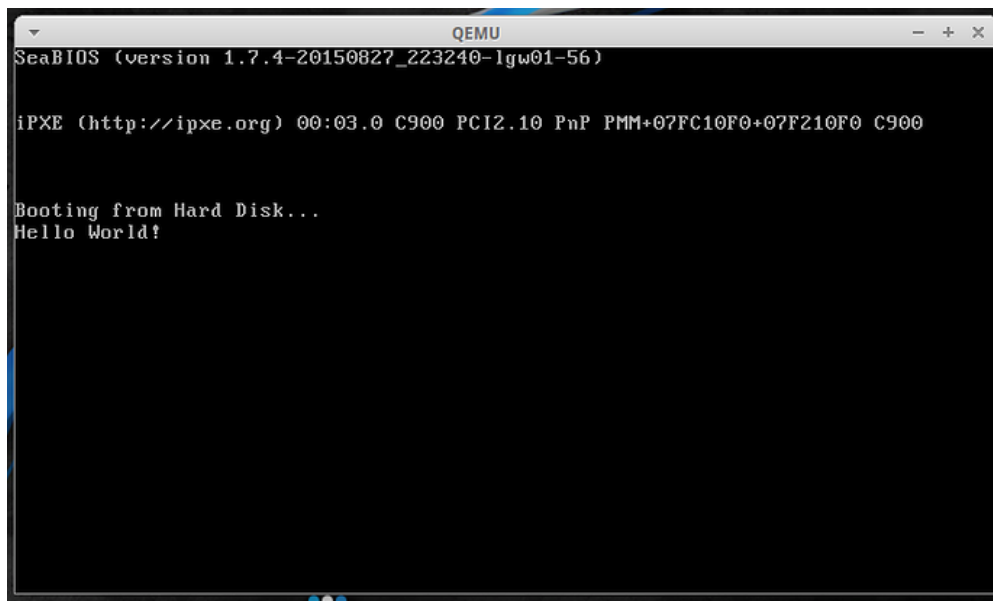
2.3 How to build and run this?

You can build it with the following commands:

```
nasm -fbin bootloader.asm -o bootloader.bin
nasm -fbin kernel.asm -o kernel.bin
cat bootloader.bin kernel.bin > result.bin
```

And run it with this:

```
qemu-system-i386 result.bin
```



3 Part 3: Building & Running

3.1 Building everything

Building a resulting binary (one that's bootable) would work either by assembling each file on it's own and put them together after that or by assembling one main file and making them include each other. The second option is obviously less work as only one file needs to be assembled, addresses and offsets are handled by the assembler and you can't accidentally copy things to the wrong place. Overall we will do both (putting together and including) as the bootloader and kernel both need different starting addresses that are not necessary subsequent which means putting them in one file / making one include the other may result in unwanted problems.

3.1.1 Installing NASM

```
sudo apt-get install nasm
```

3.1.2 Installing NASM on Windows

Download from <http://www.nasm.us/>

(Click Download, search for a recent version, click on win32/win64, download installer)

Optional: Add the installation folder to the PATH variable for quick usage of the 'nasm' command

3.1.3 Assembling with NASM:

```
nasm -fbin <file> -o <output>
```

-f is the format in this case raw binary, -o is the output file

3.1.4 Putting binaries together

```
cat file01.bin file02.bin file03.bin > result.bin
```

```
cat bootloader.bin kernel.bin > result.bin
```

puts those binaries together to one big binary in exactly this order

3.1.5 Putting binaries together on Windows

```
copy /b file01.bin + file02.bin + file03.bin result.bin
```

```
copy /b bootloader.bin + kernel.bin result.bin
```

/b indicates that this operation is about binaries.

3.1.6 Example for %include in NASM

```
..  
file1:  
%include "file01.asm"  
%include "file02.asm"  
file3:  
%include "file03.asm"  
..
```

if you now assemble this file you get a big binary as a result

it contains the code of all included files as the

preprocessor just puts the content of those files to the place you included it to

3.2 Running in Qemu and Debugging

Running our resulting bootloader and kernel binary in an emulator is pretty easy as well. Most other resources list both Qemu and Bochs, I will skip Bochs though as it's pretty slow compares to Qemu and requires more configuration but I will link an article with a "Hello World" bootloader emulated in Bochs. Qemu makes it not only possible to quickly test our build OS but also to debug it (Thanks to @hkh4cks and their MiniOS for making me look into that).

3.2.1 Installing Qemu

```
sudo apt-get install qemu-system
```

3.2.2 Installing Qemu on Windows

Download from <http://www.qemu.org/> (Click Download and look for Windows instructions)

Optional: Add the installation folder to the PATH variable

for quick usage of the 'qemu-system-i386' command

Optional: If you want to debug your OS you need gdb.

You can get this for Windows though MinGW (<http://www.mingw.org/>)

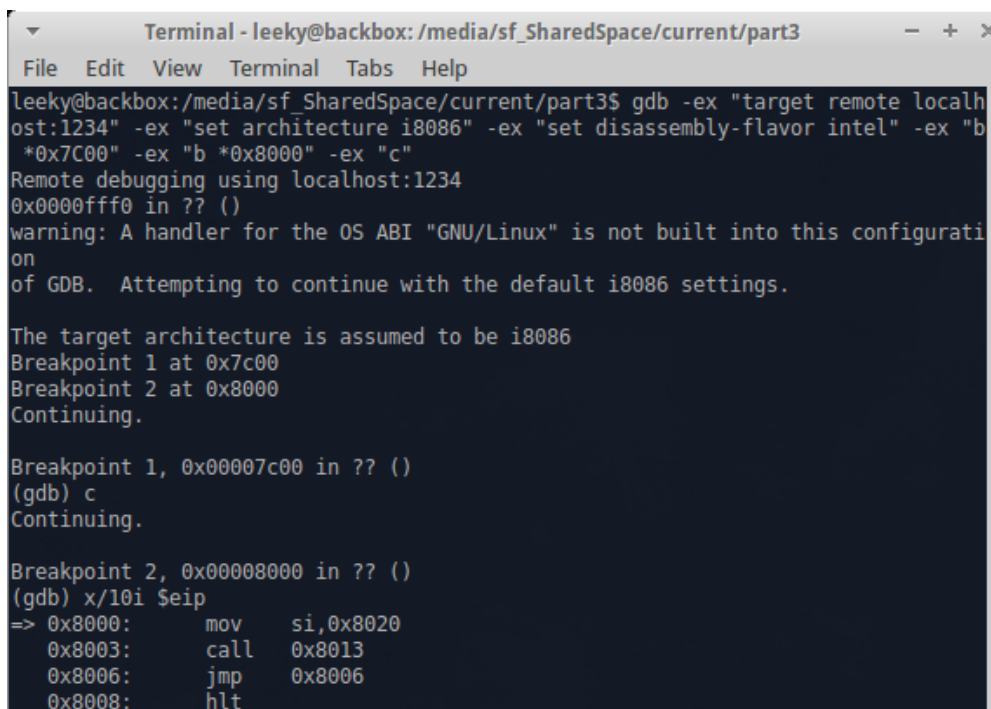
3.2.3 Running with Qemu

```
qemu-system-i386 result.bin
```

3.2.4 Debugging with Qemu

```
qemu-system-i386 -s -S result.bin
```

```
# -s starts a server on port 1234 gdb can connect to, -S makes qemu wait for gdb before executing code
gdb -ex "target remote localhost:1234"
    -ex "set architecture i8086" -ex "set disassembly-flavor intel" -ex "b *0x7C00" -ex "b *0x8000" -ex "c"
# starts gdb and executes the following commands: connect to qemu's server,
    tell gdb following code is supposed to be shown in 16bit intel assembly,
# set breakpoint at bootloader start, set breakpoint at kernel start, continue till bootloader starts
```



```
Terminal - leeky@backbox: /media/sf_SharedSpace/current/part3
leeky@backbox:/media/sf_SharedSpace/current/part3$ gdb -ex "target remote localhost:1234" -ex "set architecture i8086" -ex "set disassembly-flavor intel" -ex "b *0x7C00" -ex "b *0x8000" -ex "c"
Remote debugging using localhost:1234
0x0000ffff in ?? ()
warning: A handler for the OS ABI "GNU/Linux" is not built into this configuration of GDB. Attempting to continue with the default i8086 settings.

The target architecture is assumed to be i8086
Breakpoint 1 at 0x7c00
Breakpoint 2 at 0x8000
Continuing.

Breakpoint 1, 0x00007c00 in ?? ()
(gdb) c
Continuing.

Breakpoint 2, 0x00008000 in ?? ()
(gdb) x/10i $eip
=> 0x8000:    mov     si,0x8020
0x8003:    call    0x8013
0x8006:    jmp     0x8006
0x8008:    hlt
```

3.2.5 Difference to Real System

There are quite a bit of differences considering that qemu only emulates a real system. So it might happen quite often that code works in qemu but doesn't on a real system (or in reverse), especially when directly accessing memory and not using interrupts. This happened to me quite often when reading from the drive or working with custom interrupt handlers. Also qemu just boots the image, it doesn't check if it is in a valid format except if the MBR has the correct signature.

3.3 Running on a real system

Running this your real mode OS is most likely the coolest part! But it's also the part where the most stuff can go wrong. I've only tried to boot it from USB Flash Drives so far but it should work for SD Cards as well, I can't speak for other devices though. So before reading further I wanted to post a small list of things that I know of causing this to not work:

- Your BIOS might be configured to boot UEFI only. You can change this in your BIOS configs (press all F keys [F1,F2,...] on boot to open the configuration menu).
- Your BIOS might be configured to fast boot which makes the BIOS skip searching for devices. Just google how to change this as it depends on your main OS.
- Your BIOS might be configured to secure boot only (this is supposed to prevent malicious devices to boot). Change this in the BIOS configs.
- Device with realmode OS might be low in boot priority. Either change the boot priority within the BIOS configs or choose manually on boot (spamming F10 or F12 after power on/restart to open up the boot menu worked on the systems I tested)
- Your device doesn't contain a correct BIOS Parameter Block which might confuse the BIOS (depends on the device your realmode OS is on)
- Your BIOS doesn't allow booting from a Floppy Disk Drive (I'm sure that there are workarounds for this but I don't know how)
- I hope I don't have to mention that you need a x86 compatible CPU (x86 or x86-64) to run this on your system
- Overall everything here is as low level as it can get so there might be numerous other reasons why stuff doesn't work

Note: I'm very sorry in advance but if nothing posted here helps booting it on your system I won't be able to help you as I barely got it working my computer (after a long time of trial and error).

3.3.1 Floppy Disk Drive emulation with a USB Flash Drive

If you tried to boot the binary from the last part (I don't really expect that anyone did that) you might have noticed that even though you selected your device with OS on it, it just skipped over it and booted your normal operation system. Well at least that happened to me. You might also have noticed that your system doesn't really recognize your device (and recommends to format it - Windows). I actually don't know why systems (the computers I tested it on at least) aren't able to boot the raw USB Flash Drives (shame on me) but I know how we can "disguise" our USB Drive as a Floppy Disk Drive by adding a FAT extended Standard BIOS Parameter Block which makes the content on the drive correctly bootable (The Standard BIOS Parameter Block alone might work but adding a few bytes for the FAT extension won't hurt and tell normal OS that this is a FAT formatted USB Device which removes the "recommend to format" message. DON'T TRY TO COPY DATA ON IT THOUGH AS IT'S NOT A REAL FAT12/FAT16 IMPLEMENTATION). A BPB (BIOS Parameter Block) contains all data about the layout of the drive and how things are saved on it. I've already talked about a few of them in Part 2 (LogicalSectors, SectorsPerTrack, Sides) and I won't go into the other values any further if you are interested in more look into the reference section.

I've modified Part 2's bootloader with a modified version MikeOS's BPB and confirmed everything working on my computer:

```

org 0x7C00
;Note that the jump and NOP are part of the BPB
jmp short start
nop

; The following code wasn't written by me
; it's just a Standard BIOS Parameter Block with a FAT12/FAT16 extension
; considering the comments are pretty good and already describe the use of each value
; we might just use this as it's working (which is something I had many problems with).
; Source MikeOS
; http://mikeos.sourceforge.net/
; -----
; Disk description table, to make it a valid floppy
; Note: some of these values are hard-coded in the source!
; Values are those used by IBM for 1.44 MB, 3.5" diskette
OEMLabel      db "Example "    ; Disk label
BytesPerSector dw 512          ; Bytes per sector
SectorsPerCluster db 1        ; Sectors per cluster
ReservedForBoot dw 1          ; Reserved sectors for boot record
NumberOfFats   db 2           ; Number of copies of the FAT
RootDirEntries dw 224         ; Number of entries in root dir
; (224 * 32 = 7168 = 14 sectors to read)
LogicalSectors dw 2880        ; Number of logical sectors
MediumByte     db 0F0h        ; Medium descriptor byte
SectorsPerFat   dw 9           ; Sectors per FAT
SectorsPerTrack dw 18         ; Sectors per track (36/cylinder)
Sides          dw 2           ; Number of sides/heads
HiddenSectors   dd 0          ; Number of hidden sectors
LargeSectors    dd 0          ; Number of LBA sectors
; MikeOS's bootloader didn't mention this but the FAT12/FAT16 extension starts here
DriveNo        dw 0           ; Drive No: 0
Signature      db 41          ; Drive signature: 41 for floppy
VolumeID       dd 00000000h    ; Volume ID: any number
VolumeLabel    db "Example "  ; Volume Label: any 11 chars
FileSystem     db "FAT12 "     ; File system type: don't change!
start:
; -----

;Reset disk system
mov ah, 0
int 0x13 ; 0x13 ah=0 dl = drive number

;Read from harddrive and write to RAM
mov bx, 0x8000 ; bx = address to write the kernel to
mov al, 1      ; al = amount of sectors to read
mov ch, 0      ; cylinder/track = 0
mov dh, 0      ; head = 0
mov cl, 2      ; sector = 2
mov ah, 2      ; ah = 2: read from drive
int 0x13      ; => ah = status, al = amount read
jmp 0x8000
times 510-($-$$) db 0
;Begin MBR Signature
db 0x55 ;byte 511 = 0x55

```

```
db 0xAA ;byte 512 = 0xAA
```

3.4 Deploying to a USB Device

Ok this will be a bit awkward as I mainly work with Windows which is why I don't have much knowledge how to do this on Linux Systems. (Please don't hate me :() I also don't want to write about untested stuff here so I'll just post a few references here instead. (Basically you have to write a raw binary to the USB Drive and overwrite the sector 0 of it which isn't possible through simple copying)

3.4.1 References for doing this on Linux

- <http://f.osdev.org/viewtopic.php?f=1&t=28331>
- <https://stackoverflow.com/questions/1894843/how-can-i-put-a-compiled-boot-sector-onto-a-usb-stick-or-disk>
- <https://www.cyberciti.biz/faq/howto-copy-mbr/>
- <https://bipedu.wordpress.com/2013/06/22/live-usb-linux-with-dd-command/>
- <https://askubuntu.com/questions/22381/how-to-format-a-usb-flash-drive>

3.4.2 Writing to the USB using HHDRawCopy (Windows)

WARNING: THIS WILL OVERWRITE THE CONTENT OF YOUR USB DRIVE
AND YOU WON'T BE ABLE TO USE IT WITHOUT FORMATING IT BEFORE
Download HHDRawCopy from <http://hddguru.com/software/HDD-Raw-Copy-Tool/>
Start it
Double click the entry with the BUS FILE
Select your resulting binary
Click Continue
Select your USB Drive
Click Continue
Click Start

3.4.3 Looking at the binary in a hexeditor (Windows)

Download HxD from <https://mh-nexus.de/en/hxd/>
Either start it normally and look at the resulting binary
Or start it with administrator privileges to read directly from your USB Device to check content

3.4.4 Making your USB Usable again (Windows)

I had experience with the Windows formating tool not working so I thought I might just put this here for Windows User:

1. Open console
2. diskpart
3. list disk
4. select disk <YOUR DEVICE NUMBER>
5. clean
6. create partition primary
7. format quick fs=fat32
8. active
9. assign
10. exit

3.4.5 Booting it

Start/Restart your computer with the drive containing your realmode OS plugged in. It will now either boot automatically (because of boot priority) or you will need to select it manually by opening the boot selection menu (spamming F10/F12 on boot for me). If this doesn't work look at the "Running on a real system" section and at the reasons why it may have failed.

4 Part 4: Basic Math OS

4.1 Let's start

So this time we will write a basic math kernel. It will be able to read in commands/math operations and call functions depending on the command entered and after that it should ask for 2 numbers to do the math operation with and then print out the result.

Something like:

```
add
1
5
=> 6
```

To make our basic math operation system work we will need functions to...

1. Print out strings to tell the user what to do (already covered in part 2)
2. Read in user input (real mode code)
3. Compare input to check which command was entered (general code)
4. Convert inputted string to numbers to calculate with them (general code)
5. Calculate with those numbers
6. Print out the resulting number (real mode code)

4.2 Reading in strings

To read input from our keyboard we use the interrupt 0x16. To specify we will use interrupt 0x16 0 and 0x16,1. Interrupt 0x16 0 will make the system halt and wait until a key is pressed, it then puts the scan code of the key pressed into AH and the ASCII character (if present for the key pressed) into AL. (See references for scan code table) Sadly 0x16,0 behaves inconsistent and doesn't always return the wanted result. This can be fixed by checking if a key is pressed at the moment with int 0x16,1 which returns the last pressed scan code into AH and AL (same as 0x16,0) and sets the zero flag depending on if a key on the keyboard is currently pressed:

int	ah	Zero flag	Returned al	Returned ah	Description
int 0x16	0x0	Unchanged	Scan code	ASCII Code	Halts until key is pressed
int 0x16	0x1	No Key pressed	Scan code	ASCII Code	Read Keyboard status

```
;-----
;=>al = character read(ascii) => ah = character read(keycode) => zf = (1 if nothing read) /
readChar:
    mov ah, 1; int 0x16, 0 should be enough but it behaves inconsistent without 0x16, 1 before it
    int 0x16; int 0x16, 1 => check if a key is pressed (set zero flag if not)
    jz .end
    mov ah, 0; int 0x16, 0 => halts till key is pressed and returns it into al and ah
    int 0x16
    ret
.end:
    mov ax, 0; if no character was pressed return al = 0 and ah = 0
    ret
;-----
```

To read in strings we will just loop through readChar and save the resulting characters into a temporary buffer. We will have to watch out for some exceptions though, for example backspace (removing a character from the buffer) and enter (returning so far entered string). Also we have to watch out to not forget to null-terminate the string. The following code has exceptions for backspace and enter but doesn't provide live feedback (doesn't print out the characters entered while typing them), look at the Github repository for a version with it included:

```

;-----
;=>di = string inputted => zf = (1 if nothing read) | changes ax
#define readString_size 8 ;maximum amount of characters

readString:
    mov di, .buffer;stosb writes into di so let us set di to the start of the buffer
    .inner:
        call readChar;read a character
        jz .inner;if input == 0 repeat reading until enter pressed or max size reached
        cmp ah, 0x1C;zeroflag if enter pressed
        je .end
        cmp ah, 0x0E;zeroflag if backspace pressed
        je .remove
        stosb;store character into buffer and increase di
        cmp di, (.buffer+readString_size) ;if length of di is >= readString_size => end
        jge .end
    jmp .inner;read next character
    .remove:
        cmp di, .buffer;if di is at index 0 do not remove character as it is already at the beginning
        jle .inner
        dec di;remove a character by moving one index back
        jmp .inner
    .end:
        xor al, al
        stosb ;zero terminate string by adding a \0 to it
        mov di, .buffer;set output to buffer beginning
        ret
    .buffer resb (readString_size+1)
;-----

```

4.3 Comparing strings

Comparing strings is essential to be able to process multiple different commands and to differentiate between them. We do this by first calculating the length of one of the strings and then checking if all characters in the range are equal. It doesn't matter which of the strings we measure for the length as the zero terminator will be taken as part of the string which means if the zero terminator isn't at the same spot they won't be thought as equal.

```

;-----
;zero flag = set if equals |
stringCompare:
    pusha;save all registers
    or cx, -1;set cx to biggest unsigned number
    xor al, al;set al to 0
    repne scasb;scan through di until zero terminator was hit and decrease cx for each scanned character
    neg cx;calculate length of di by negating cx which returns the length of the string including zero terminator
    sub di, cx;reset di by setting it to the original index it started with
    inc di
    repe cmpsb;check if character from di match with si (including zero terminator)
    test cx, cx;test if amount of matching = size of string, set zero flag if equals
    popa;restore all registers
    ret
;-----

```

4.4 Converting strings to numbers

So now we can print and read characters and strings but how we write and read numbers? Well, we have to convert the numbers to ascii-characters and the ascii-characters to numbers. The method of doing this depends on the number base (2, 8, 10, 16) we work with. The code for this one is written for base 16/hexadecimal numbers but the theory behind it works for base N.

Converting a hex string into a number works by reading character after character and multiplying the previous result with 16 before adding the value of the last read character. This is based on how

numbers representation works with the Nst digit of a number being actually $\text{Nst digit} * \text{base}^N$ which means this works with base N by replacing the number to multiply with with the base.

Example:

A	= 10	= 10
AB	= $10 * 16 + 11$	= 171
ABF	= $(10 * 16 + 11) * 16 + 15$	= 2751
ABFF	= $((10 * 16 + 11) * 16 + 15) * 16 + 15$	= 44031
A	= A	= A
AB	= $A * 10 + B$	= AB
ABF	= $(A * 10 + B) * 10 + F$	= ABF
ABFF	= $((A * 10 + B) * 10 + F) * 10 + F$	= ABFF

```

;-----
;=> dx = hex => zf (zf = FAILED), si = input str /
hexstr2num:
    push ax;save ax and si
    push si
    xor dx, dx;reset dx and ax to use them for working (dx will contain the resulting number)
    xor ax, ax
    .loop:
        lodsb;load ASCII character from inputted string SI into AL and increase SI
        test al, al;end reading in if reached zero terminator
        jz .end
        shl dx, 4;shifting left by 4 => multiplying with 16
        cmp al, '0'
        jl .error;if character is less than 0x30 it can not be a number or character
        cmp al, '9'
        jle .num;if character is within the range of 0x30 and 0x39 it is a number
        cmp al, 'A'
        jl .error;if character is bigger than 0x39 and smaller than 0x42 it is not a character
        cmp al, 'F'
        jle .clet;if character is within the range of 0x42 and 0x46 it is a uppercased hex character
        jmp .error;if it is not a number or a hex character => error
    .num:
        sub al, '0';subtract 0x30 from ASCII number to get value
        jmp .continue
    .clet:
        sub al, 'A'-0xA;subtract 0x42 and add 0xA to ASCII uppercased hex character to get value
    .continue:
        add dx, ax;lastResult = (lastResult * 16) + currentNumber;
        jmp .loop;loop to the next character
    xor ax, ax
    cmp ax, 1;ax != 1 => zero flag not set
    jmp .end
    .error:
        xor dx, dx
        test dx, dx;dx == 0 => zero flag set
    .end:
    pop si;restore si and ax
    pop ax
    ret
;-----

```

4.5 Converting numbers to strings

Converting a number to a string (in this case a hex string) works by printing out the digits its made of one by one. The code I used prints out the content of a 8bit register which means 2 hex digits fit in it. The first 4bit and the last 4bit can both be seen as independent numbers and both represent one of the 2 digits the 8bit register will be displayed as. For a hexadecimal number we need to take in account that digits with a value of 0-9 are to be displayed as a number and digits with a value of 10-15 as letters (A,B,C,D,E,F).


```

;-----
;dl = value to print out /
printHex:
    call .print;call twice to print first 4 bit and last 4 bit
.print:
    ror dl, 4;because we want to print last 4 bits first we rotate
    pusha;save registers (as dl has 8bit and gets rotated twice it will end up unchanged as well)
    and dl, 0x0F;only read first 4 bits into dl
    mov al, ('0'+0xA);NUM BASE = '0' if dl < 10
    mov bl, 'A';NUM BASE = 'A' if dl >= 10
    sub dl, 0xA;subtract 10 from dl to set flags
    cmovae ax, bx;if dl >= 0xA: set NUM BASE = 'A'
    add al, dl;dl+NUM BASE = ASCII Character
    call printChar;prints ASCII character in al
    popa
    ret
;-----

```

4.6 Screenshot of it working

```

QEMU - Press Ctrl+Alt+G to release grab
Machine View
Input: add
[11]: A
[21]: C
[R1]: 0016
Input: sub
[11]: EF
[21]: F
[R1]: 00E0
Input: mul
[11]: 10
[21]: 2
[R1]: 0020
Input: div
[11]: 3000
[21]: 3
[R1]: 1000
Input: help
Known commands: help, sub, add, mul, div and clear
Input: _

```

5 Part 5: Graphic Mode

5.1 How to render in real mode

You might have asked yourself how the BIOS was able to render text as shown in the previous parts. Well the answer to that is also the answer of how to render more custom images/shapes (as it uses the same interrupt routines): After the system BIOS (the one installed on the motherboard) is done with its basic initializing it starts loading the Video BIOS (installed on the graphic card/chip set) which then handles interaction screen interactions through BIOS routines.

By default the Video BIOS starts in a Black/White mode with 40 characters for 25 lines (Mode 00).

As you might remember the interrupt to print characters we used (interrupt 0x10,0xE) had a color attribute not usable for us up to now because of the color restriction.

int	ah	al	bh	bl	Description
0x10	0xE	ASCII Character	Page to write to	Color Attribute	Print a character to the screen

By changing the display mode to Mode 01 we would get access to 16 colors with the same 40 characters for 25 lines configuration and could actually specify colors for text output.

But as you might have guessed we won't use a text mode (like Mode 00 or 01) but a mode with graphic output. There are a few available and all of them have unique characteristics but from my experience Mode 0x13 is the easiest to use by both interrupts and direct memory access and it provides a good screen size of 320x200 pixel and can display 256 colors at the same time (color palette can be changed through interrupts). Reason for it being easy to use is that unlike other VGA modes it doesn't use a rather complex reference mode but is accessible as a plain memory with each byte mapped to a pixel on screen.

(More information about the specifications of the BIOS display modes within the "VGABIOS.TXT" and "INTERRUP.TXT") (More information about how the different graphic modes work at the "OS-DevVga" resource)

We can switch graphic modes by using interrupt 0x10, 0x0 and setting al to the display mode we want.

int	ah	al	Description
0x10	0x13	Number representing new Display Mode	Sets display mode

If we are in a graphic mode (not a text mode) we can use the interrupt 0x10, 0xC to draw pixels.

int	ah	al	bh	cx	dx	Description
0x10	0xC	Index of color	Page to write to	x-position	y-position	Draws a pixel

As I already said we can also write pixels by writing directly to RAM which is significantly faster especially within an emulator but the method to write and the address to write to change for each graphic mode which makes it quite complex to generalize. We will look into that in a later article (but it will be limited to graphic mode 0x13).

Also note that we can still print text within a graphic mode which is really useful for debugging purpose.

OK, so we will use display mode 0x13 so let's switch to it:

```

mov ah, 0    ;Set display mode
mov al, 13h  ;13h = 320x200, 256 colors
int 0x10     ;Video BIOS Services

```

5.2 Drawing squares

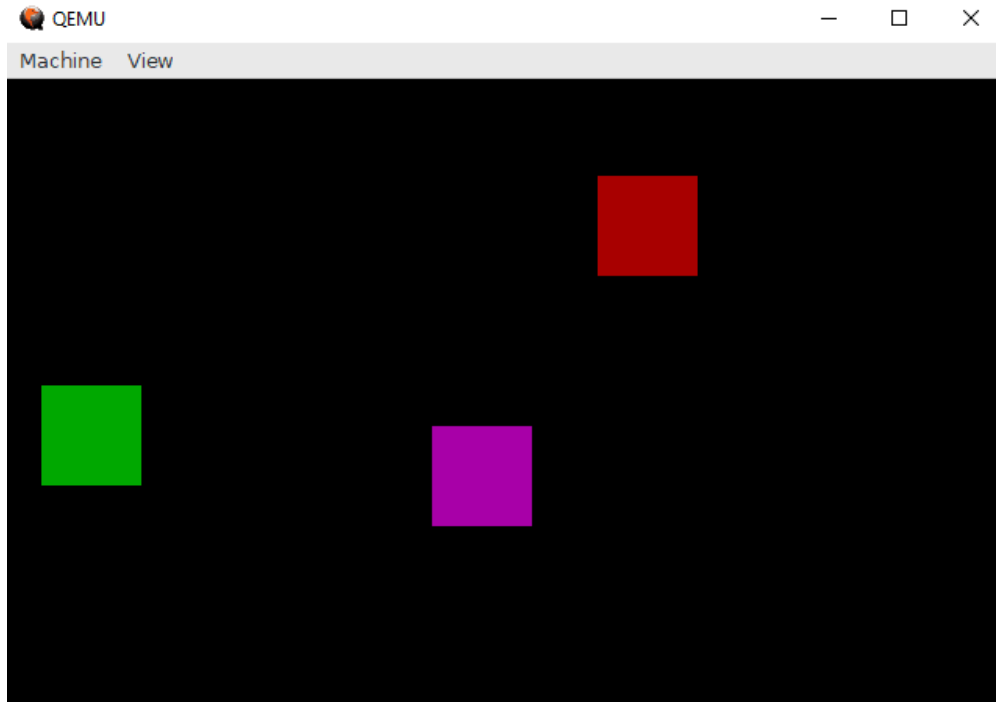
So we know that we can draw single pixels through interrupt 0x10, 0xC but how about more complex shapes? Let's start simple with a function to draw a square:

Drawing a square consist of iterating through a x-axis and a y-axis and drawing a pixel for each position.

```

;-----
;cx = xpos , dx = ypos, si = x-length, di = y-length, al = color
drawBox:
    push si                ;save x-length
    .for_x:
        push di            ;save y-length
        .for_y:
            pusha
            mov bh, 0       ;page number (0 is default)
            add cx, si      ;cx = x-coordinate
            add dx, di      ;dx = y-coordinate
            mov ah, 0xC     ;write pixel at coordinate
            int 0x10        ;draw pixel!
            popa
            sub di, 1       ;decrease di by one and set flags
            jnz .for_y      ;repeat for y-length times
            pop di          ;restore di to y-length
            sub si, 1       ;decrease si by one and set flags
            jnz .for_x      ;repeat for x-length times
            pop si          ;restore si to x-length -> starting state restored
        ret
;-----

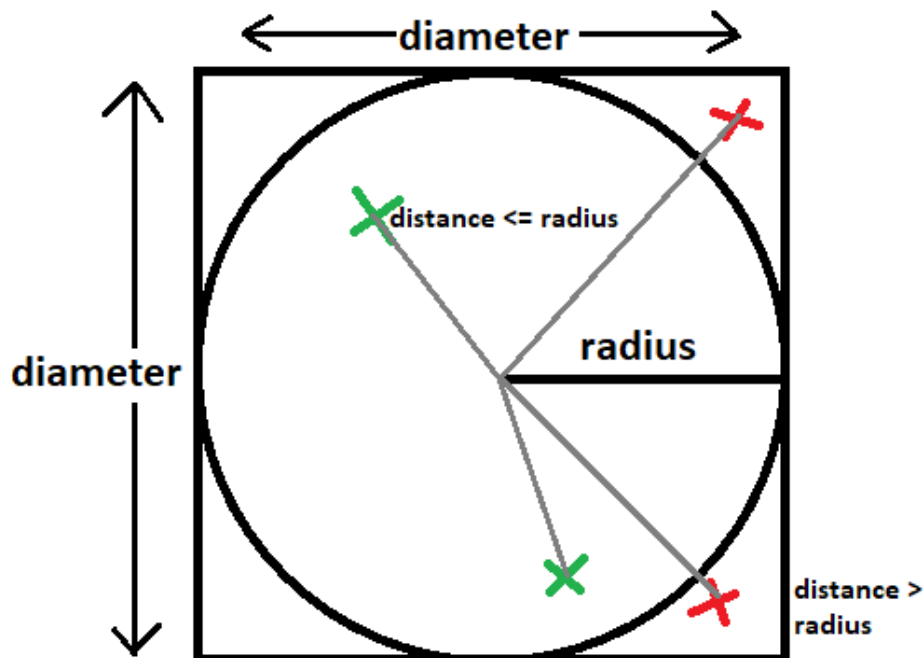
```



5.3 Drawing circles

OK, drawing a square was quite simple but how about a circle?

Drawing circles works by iterating through x- and y-axis and then only drawing the pixels with a distance from the middle point lower or equal to the radius.



```

x, y = position of pixel
middle = middle of circle
distance = sqrt((x-middle.x)^2 + (y-middle.y)^2)
if distance <= radius:
    draw pixel

```

Now you might notice the square root function within the pseudo code and although using the FPU(FSQRT) would make that possible a more cheaty approach is way simpler:

```

    if distance <= radius:
        draw pixel

<=>  if distance^2 <= radius^2:
        draw pixel

<=>  distanceNSQRT = (x-middle.x)^2 + (y-middle.y)^2
    if distanceNSQRT <= radius^2:
        draw pixel

;-----
;cx = xpos , dx = ypos, si = radius, al = color
drawCircle:
    pusha                                ;save all registers
    mov di, si
    add di, si                            ;di = si * 2, di = diameter y-axis
    mov bp, si                            ;bp = copy of radius
    add si, si                            ;si = si * 2, si = diameter x-axis
    .for_x:
        push di                            ;save y-length
        .for_y:
            pusha
            add cx, si                    ;cx = x-coordinate (start x + si)
            add dx, di                    ;dx = y-coordinate (start y + di)
                                           ;(x-middle.x)^2 + (y-middle.y)^2 <= radius^2
                                           ;(si-bp)^2 + (di-bp)^2 <= bp^2
            sub si, bp                    ;di = y - r
            sub di, bp                    ;di = x - r
            imul si, si                    ;si = x^2
            imul di, di                    ;di = y^2
            add si, di                    ;add (x-r)^2 and (y-r)^2
            imul bp, bp                    ;signed multiplication, r * r = r^2
            cmp si, bp                    ;if r^2 >= distance^2: point is within circle
            jg .skip                       ;if greater: point is not within circle
            mov bh, 0                      ;page number (0 is default)
            mov ah, 0xC                    ;write pixel at coordinate
            int 0x10                       ;draw pixel!
            .skip:
            popa
            sub di, 1                      ;decrease di by one and set flags
            jnz .for_y                    ;repeat for y-length
            pop di                          ;restore di
            sub si, 1                      ;decrease si by one and set flags
            jnz .for_x                    ;repeat for x-length
            popa                            ;restore all registers
        ret
;-----

```



5.4 Drawing an image

By default the color indexes of interrupt 0x10, 0xC refer to the VGA 256-color palette and although those are changeable through interrupts we can see already see that 256 colors are quite a bit (enough to make the color black appear 10 times in it).



Let's try to encode a small image into those indexes and display it!

```
from PIL import Image #Import Image from Pillow

paletteFile = "colors.png" #palette the BIOS uses
convertFile = "fox.png" #image to turn into a binary
outputFile = "image.bin" #name of output file

pal = Image.open(paletteFile).convert('RGB')
palette = pal.load() #load pixels of the palette
image = Image.open(convertFile).convert('RGB')
pixels = image.load() #load pixels of the image

binary = open(outputFile, "wb") #clear/create binary file

list = [] #create a list for the palette
for y in range(pal.height):
    for x in range(pal.width):
        list.append(palette[x,y]) #save the palette into an array

#write width and height as the first two bytes for variable image size
binary.write(bytearray([image.width&0xFF,image.height&0xFF]))
data = []
for x in range(image.width):
    x = image.width - x - 1 #invert x-axis (for shorter assembly code)
    for y in range(image.height):
        y = image.height - y - 1 #invert y-axis (for shorter assembly code)
        difference = 0xFFFFFFFF #init difference with a high value
```

```

#the index of the color nearest to the original pixel color
choice = 0
index = 0                                     #current index within the palette array
for c in list:
    dif = sum([(pixels[x,y][i] - c[i])**2 for i in range(3)]) #calculate difference for RGB values
    if dif < difference:
        difference = dif
        choice = index
    index += 1
#write nearest palette index into binary file as 1 byte
binary.write(bytearray([choice&0xFF]))
binary.close()                               #close file handle
print("Done.")

```

```

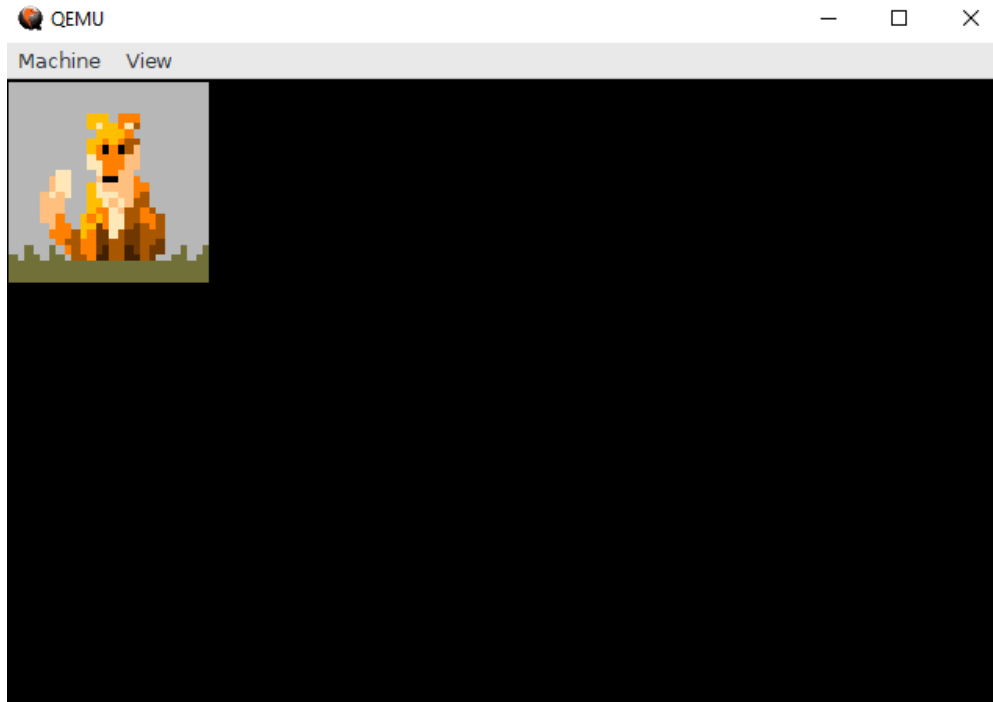
;-----
;si = image source
drawImage:
    pusha
    xor ax, ax
    lodsb
    mov cx, ax                                ;x-length (first byte of binary)
    lodsb
    mov dx, ax                                ;y-length (2nd byte of binary)
    .for_x:
        push dx
        .for_y:
            mov bh, 0                          ;page number (0 is default)
            lodsb                               ;read byte to al (color) -> next byte
            mov ah, 0xC                         ;write pixel at coordinate (cx, dx)
            int 0x10                            ;draw!
            sub dx, 1                           ;decrease dx by one and set flags
            jnz .for_y                          ;repeat for y-length
            pop dx                              ;restore dx (y-length)
        sub cx, 1                              ;decrease si by one and set flags
        jnz .for_x                             ;repeat for x-length
    popa
    ret
;-----

```

```

imageFile: incbin "image.bin"                ;include the raw image binary into the kernel

```

As you might notice the script tries to find the most fitting palette color for a given pixel, this isn't lossless and thus changes the image appearance and quality (obviously considering we just encoded 3 bytes (RGB8) into 1 byte (index within a palette)). We will look into ways to load our own palettes in a later part to display colors correctly (at least if you don't intent to display more than 256 unique colors at the same time).

5.5 Problems with rendering using interrupts

As of now we've only rendered static images and everything was fine but for something like a game we would want our screen to be more dynamic and render different things each frame. As we don't want previous frames to overlap with our current one we will need to clear the screen before rendering the next frame and that's where stuff goes wrong.

*No code included as it's nothing new really, look into the git repository if you are interested

[Link to Video as GIFs are not displayable in PDF Format](#)

As you might see in this video the movement of the square is slow and sometimes the screen flickers black. That is because clearing the screen (drawing a black box that covers the screen) is heavy on the emulator and slows it down remarkable. You most likely won't notice this on real hardware as the interrupts are processed way faster but this throws up the problem of timing and synchronization. For example long render time could mess up game logic processing or cause the game to run on different tick rates depending on hardware (which isn't good). And now this annoying flacking (it will also happen on real hardware): The screen refreshes at a specific rate and timing and because our rendering code is not synchronized to that the screen sometimes gets displayed even though the rendering code isn't fully executed yet. Those and a few other problems (and how to solve them) will be the topic in the next few parts!

6 Part 6: Let's write a small game (1/2)

6.1 Let's start

So let's start with listing what we essentially need for our game: We need graphical rendering and a controllable character.

6.2 Graphical Rendering

As already said the graphical rendering in this part will be done with direct memory access instead of using interrupts. For graphic mode 0x13 this is actually pretty straightforward as we can write the color (or better said the index within the palette for a color) to the address of the pixel where all pixels displayed are directly after each other. To do this we have to write to the Video memory which is positioned at 0xA0000. For example the pixel(0,0) is at 0xA0000 the pixel(1,0) is at 0xA0001 and the pixel(0,1) is at 0xA0140 (as you might remember that mode 0x13 is 320*200 in size). In some memory modes writing onto the screen is as easy as this while in others (especially bigger VGA modes) memory is divided into several planes which makes it more complex.

In actual usage it's not possible to specify the address to 0xA0000,0xA0001,etc though because an address has to be 16bit in Realmode.

We avoid this restriction by looking into how physical addresses are calculated:

```
Logical address = A:B  
Physical address = (A * 0x10) + B  
A = 64k segment  
B = Offset within segment
```

The segment A is specified by the segment register B (CS,DS,SS,ES,FS,GS) used by the instruction that accesses memory. If we don't specify segment registers the default values for a given instruction are used, so we use segment registers for every memory interaction.

If we now for example look at the description of the STOSB instruction("Store AL at address ES:(E)DI") we can see this instruction uses the ES segment register to specify the output segment.

If we now write something like:

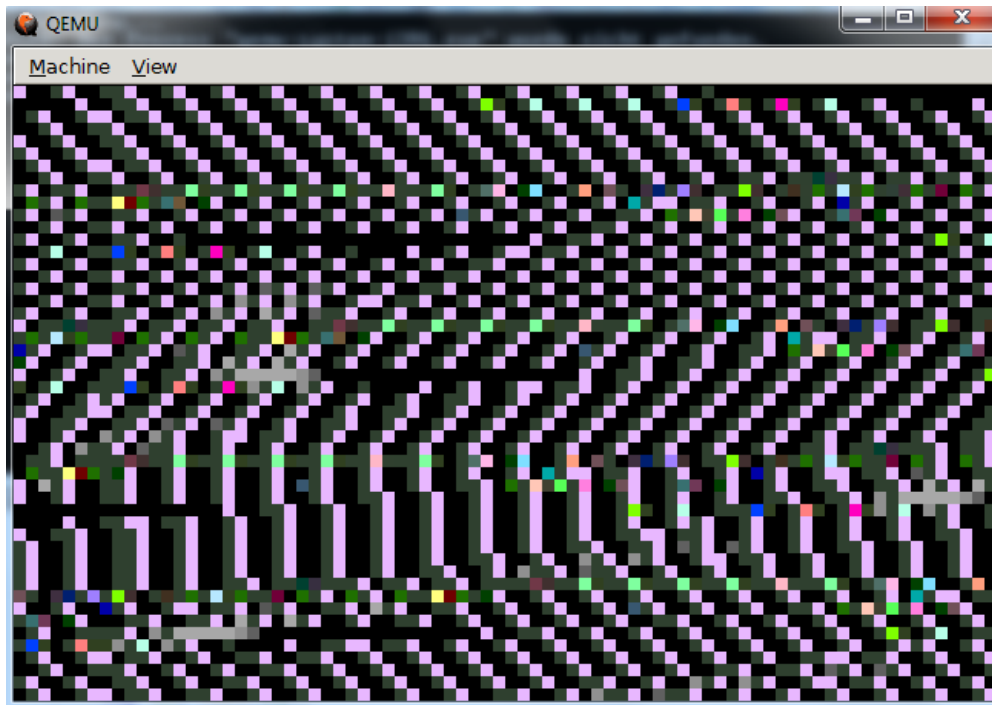
```
mov es, 0xA000 ; 0xA0000 / 0x10  
mov di, 0x140  
mov al, 0x9  
stosb
```

We write the number 9 to the physical address 0xA0140 ($0xA000 * 0x10 + 0x140 = 0xA0140 \Rightarrow \text{pixel}(0,1)$) in a pretty efficient manner.

Maybe you also remember the flicking (or better said, moments on which screen display and screen write overlapped) we had in the last part, let's also fix that by double buffering the screen. We do that by directing all our draw functions to a buffer memory region and after all drawing has been done (end of a game tick) we write the full buffer to the Video memory at once. As the screen isn't edited while game logic and rendering takes place (which might take some time) but after everything is done in a quick write we don't have synchronization problems anymore. But now we encounter a new problem, where do we get a $320 \times 200 = 64000$ byte buffer (or better said a whole segment) from? My solution isn't really ideal, I just upscaled all pixels by 4 which reduced the needed buffer size to $8050 = 4000$ which I found just somewhere within the usable

RAM. 0x7E00-0x7FFFF provides 480.5 KiB so I could have just put it into there but I liked the look of x4 pixels and it also keeps resource size small so I went with that. (not to mention that it saves me the trouble of animating a 36*48 sprite just to fill the screen the same amount)

Last thing to note before showing the actual code is that we have relative positioned sprites in the game (relative to the player) so we have to make sure that no drawing function tries to write outside our designated buffer area (which of course is limited in size) which is something that might happen if positions get negative, go outside of screen or go only partly outside of the screen. If we don't watch out for that stuff like that his happens:



```
;screen has size 320x200 but buffer only 80x50
copyBufferOver:
    pusha                    ;save all general purpose registers
    push es                 ;save segment register es
    mov es, word [graphicMemory] ;set es to the segment 0xA0000
    xor di, di
    mov cx, 200             ;for y-length of video memory
.loop:
    mov dx, cx              ;save current y position in dx
    mov cx, 320/4           ;for x-length of video memory
    .innerloop:
        mov si, 320
        sub si, cx          ;set x-offset for buffer with inverted x-axis
        mov bx, 200
        sub bx, dx          ;invert y-axis
        shr bx, 2           ;divide y position within screen by 4 to get index within buffer
        imul bx, 80         ;multiply with buffer width to get an index
        add si, bx          ;add y-offset for buffer
        add si, [screenPos] ;make the offset a logical address
        lodsb               ;read from buffer (ds:si)
        mov ah, al          ;duplicate pixel
        stosw               ;write 2*2 pixel row to graphic memory (es:di)
        stosw
    loop .innerloop
    mov cx, dx              ;restore saved y position
loop .loop
pop es                     ;restore segment register es
```

```

    popa                                ;restore all general purpose registers
    ret

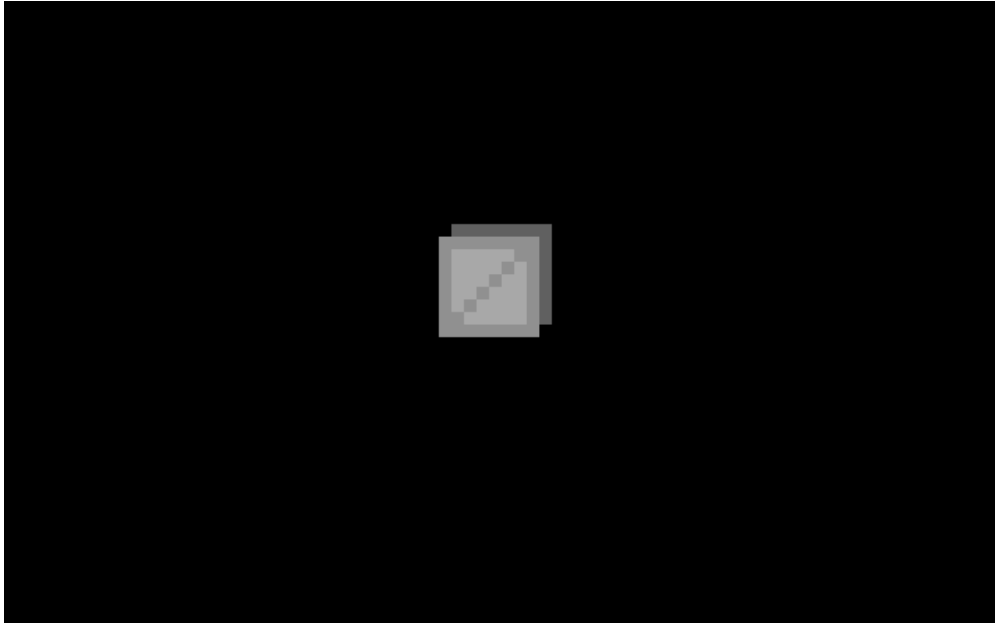
;si = position of image, ax = xpos, bx = ypos
;a bit messy because of all the error checks to not draw out of screen
drawImage:
    pusha
    xor di, di
    imul di, bx, 80                    ;set di to the y-offset*width to address an index
    add di, [screenPos]                ;add address of the buffer to make it an address within it
    add di, ax                          ;add offset of x-position
    mov bp, ax                          ;bp = x-position
    xor ax, ax                          ;reset ax (especially higher byte as it doesn't get written to by loadsb)
    lodsb                               ;read x-size of binary
    mov cx, ax                          ;save x-size into cx
    lodsb                               ;read y-size of binary
    mov dx, ax                          ;save y-size into dx
    .for_y:
        ;bx is set to the offsetOnScreen and a few checks are made to ensure the line is on screen
        mov bx, di
        add bx, cx                      ;bx = offsetOnScreen + xsize
        ;skip if line (or better said last pixel of the line) is out of screen (above screen)
        sub bx, word [screenPos]

        jl .skip_x
        sub bx, cx                      ;reset bx to the offsetOnScreen as the line appears to be on screen
        ;skip if line (or better said first pixel of the line) is out of screen (bellow screen)
        sub bx, 80*50
        jge .skip_x
        jge .skip_x
        xor bx, bx                      ;everything is fine with this line!
    .for_x:
        mov al, byte [si+bx]            ;read pixel within the image
        test al, al                     ;skip if pixel value is 0 and treat it as transparent
        jz .skip
        cmp bx, 80                      ;if pixel is right out of screen, skip it
        jge .skip
        cmp bx, 0                        ;if pixel is left out of screen, skip it
        jl .skip
        mov byte [di+bx], al            ;write pixel value from image to buffer
        .skip:
        inc bx                           ;increase index within image and buffer
        cmp bx, cx                       ;repeat until the line ends
    jl .for_x
    .skip_x:
        add di, 80                       ;next row within buffer
        add si, cx                       ;next row within image
        dec dx                           ;decrease index of line
    jnz .for_y                           ;repeat for y-length
    popa
    ret

graphicMemory dw 0xA000                ;segment register value for Video memory
screenPos dw 0x0500                    ;buffer will be at this address

```

With a modified image2binary.py script we can now finally display images:



6.3 Adding a Character

With our modified rendering code adding a player (kernel.asm@line 19) and making the objects relative to it (kernel.asm@drawEntity) isn't really a problem, we just have to subtract the player position from the other objects and then add the position of the center of the screen (because that's where the player is actually displayed).



We already talked about key input in Part 4, I used the same readChar code for this project for now:

```
checkKey:
    mov ah, 1
    int 0x16    ;interrupt 0x16 ah=1 => read key status, zeroflag if no key pressed
    jz .end
```

```
mov ax, 0
int 0x16    ;interrupt 0x16 ah=0 => read key pressed, fills al and ah
ret
.end:
mov ax, 0    ;return 0 if no key is pressed
ret
```

The key pressed decided what direction to move to, no key or a key not included in the controls results in stopping. (kernel.asm@gameControls) This way sadly has the problem of having a delay after releasing a key (or pressing a new key) which results in a delayed movement, we will fix that in the next part.

I also added collision but that's just code that checks if the hitboxes of two entities overlap so nothing special (kernel.asm@checkForCollision).

[Link to Video as GIFs are not displayable in PDF Format](#)

7 Part 7: Let's write a small game (2/2)

7.1 Interrupts you say?

Interrupts are signals raised from the CPU or from external Hardware (Keyboard, Mouse) that tell the CPU to save the current execution context, stop doing what it was doing and give execution to the handler for the specific event. Exceptions are as one example handled in that way in Realmode, but also keydown/keyup events from a keyboard get send to the CPU in this way to notify it of changes. In Realmode the interrupt handler addresses are stored in the Interrupt Vector Table (contrary to the Interrupt Descriptor Table used in protected mode) which is usually positioned at 0000:000 for comp ability reasons. Each entry in the Interrupt Vector Table (IVT) is 4 byte in size where the first two byte are representing the segment of the handler and the last two byte the offset from there to the handler, because of this very simple format we can find the position of the handler address by multiplying the interrupt number with 4 where we can then overwrite the segment:offset structure there to point to custom handler.

7.2 Writing a custom key event handler

Based on the simple structure of the IVT (and a handy IVT mapping from osdev) we can overwrite the keyboard interrupt handler pretty easily

IVT Offset	INT #	IRQ #	Description
0x0020	0x08	0	PIT
0x0024	0x09	1	Keyboard <-- We will overwrite this
0x0028	0x0A	2	8259A slave controller
0x002C	0x0B	3	COM2 / COM4
0x0030	0x0C	4	COM1 / COM3
0x0034	0x0D	5	LPT2
0x0038	0x0E	6	Floppy controller
0x003C	0x0F	7	LPT1

```
registerInterruptHandlers:
    mov [0x0024], dword keyboardINTListener ;implements keyboardListener
    ret
```

So now we will get all interrupts triggered by the keyboard (or through a software interrupt 9) in our keyboardINTListener function. Now let's look into implementing the actual handler function:

First of all we have to remember that this event is asynchronous to the other tasks the CPU does, so when entering our function we could (and will) have arbitrary values (from the functions that were executed previously) in not only our normal working registers but also in the segment registers which is why we can't assume any predictable values in them. Also it's important to know that after the interrupt handler is ended with a IRET instruction the execution will continue at the previous position with the current register values which means we also have to watch out to reset everything to the state it was before calling our handler to prevent crashes and unexpected behaviour.

Now looking at the keyboard handler itself we have to make sure that we tell the Programmable Interrupt Controller (PIC) that passed the signal through from the keyboard that it arrived correctly at the end of our code to assure that it keeps sending us future interrupts.

```
mov al, 20h ;20h
out 20h, al ;acknowledge the interrupt so further interrupts can be handled again
```

As you can see in this code snippet we do this by sending a 0x20 through the IO port 0x20. Depending on the interrupt it might be needed to send this signal to a slave PIC as well.

Reading the current key status itself is rather simple as we directly pull it from the IO port 0x60 which is the PS/2 Keyboard and Mouse Port (Don't worry because of the way Keyboards are handled on modern systems a USB Keyboard will be treated as a PS/2 here). Note here that the key mappings are not in ASCII or similar format but have their own (Link for file containing the mapping in the References section).

Now the last part we can't forget about is that because of the asynchronous nature of interrupts we can't move the player in the handler if we somehow want it synchronous to game and render ticks which means we have to save the state from the handler in memory which then can be read synchronously.

```
pressA db 0
pressD db 0
pressW db 0
pressS db 0
keyboardINTListener: ;interrupt handler for keyboard events
    pusha
    xor bx,bx ; bx = 0: signify key down event
    inc bx
    in al,0x60 ;get input to AX, 0x60 = ps/2 first port for keyboard
    btr ax, 7 ;al now contains the key code without key pressed flag, also carry flag set if key up event
    jnc .keyDown
    dec bx ; bx = 1: key up event
    .keyDown:
    cmp al,0x1e ;a
    jne .check1
    ;use cs overwrite because we don't know where the data segment might point to
    mov byte [cs:pressA], bl
    .check1:
    cmp al,0x20 ;d
    jne .check2
    mov byte [cs:pressD], bl
    .check2:
    cmp al,0x11 ;w
    jne .check3
    mov byte [cs:pressW], bl
    .check3:
    cmp al,0x1f ;s
    jne .check4
    mov byte [cs:pressS], bl
    .check4:
    mov al, 20h ;20h
    out 20h, al ;acknowledge the interrupt so further interrupts can be handled again
    popa ;resume state to not modify something by accident
    iret ;return from an interrupt routine
```

After implementing this (and a few more things) a new problem appeared which already existed when booting the kernel on a real system instead of an emulator:

[Link to Video as GIFs are not displayable in PDF Format](#)

Because we don't have any limitations on the game speed it runs as fast as the processor can, which makes movement quite troublesome.

7.3 Sleeping for a fixed amount of time

To actually maneuver in this now even in the emulator fast environment we need to slow it down. Because the game speed should be similar on different devices (and emulators) it makes sense to use a BIOS provided interrupt for that.

int	ah	cx	dx	CF	Description
int 0x15	0x68	High Word	Low Word	Set if Error	Wait for CX:DX Microseconds

synchronize:

```

pusha
    mov si, 20 ; si = time in ms
    mov dx, si
    mov cx, si
    shr cx, 6
    shl dx, 10
    mov ah, 86h
    int 15h ;cx,dx sleep time in microseconds - cx = high word, dx = low word
popa
ret

```

Now after having this sleep time in the code it works at acceptable speed both in the emulator and on a real system

[Link to Video as GIFs are not displayable in PDF Format](#)

8 Final Conclusion

So now after after 7 parts we went over the theory of how the booting process works, to basic text output, input and processing and ended with a small bootable “game” runnable on most x86 systems. Content wise there isn’t much to it, just running around and collecting coins but then again that’s just a small example of what you can do with not too much effort. I know that the last parts were getting released pretty delayed from each other but I still hope that this series was somewhat interesting and motivating to look into the lower level inner workings of operation systems and programs in general. If you any mistakes please point them out to me so I can correct them, also again feedback to the series is appreciated :)

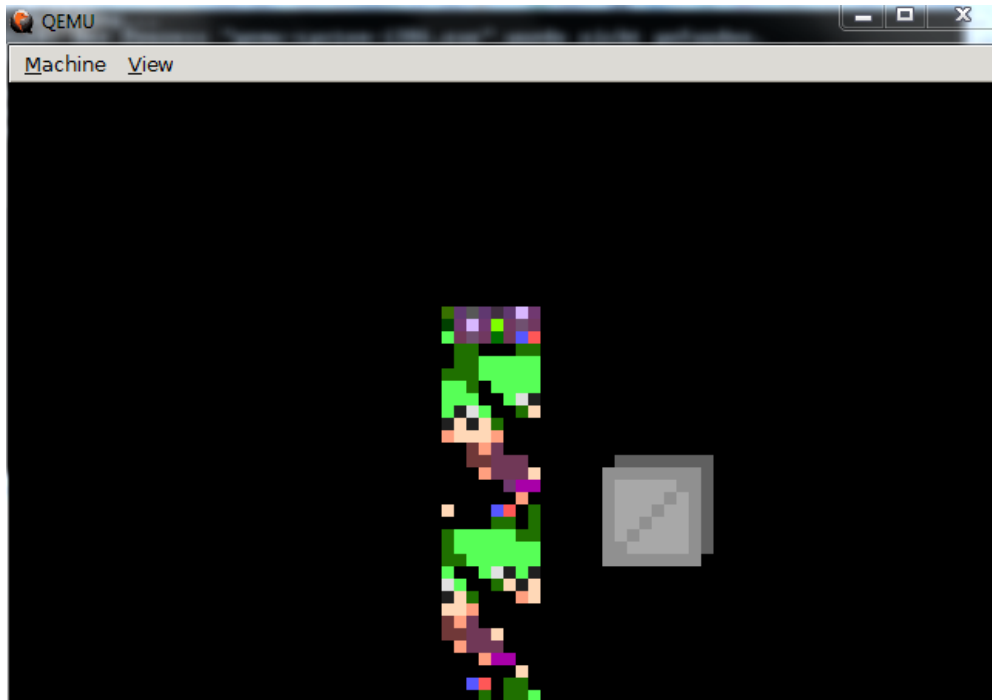
Thank you for taking your time to read through or parts of my series!

8.1 Some Development Screenshots

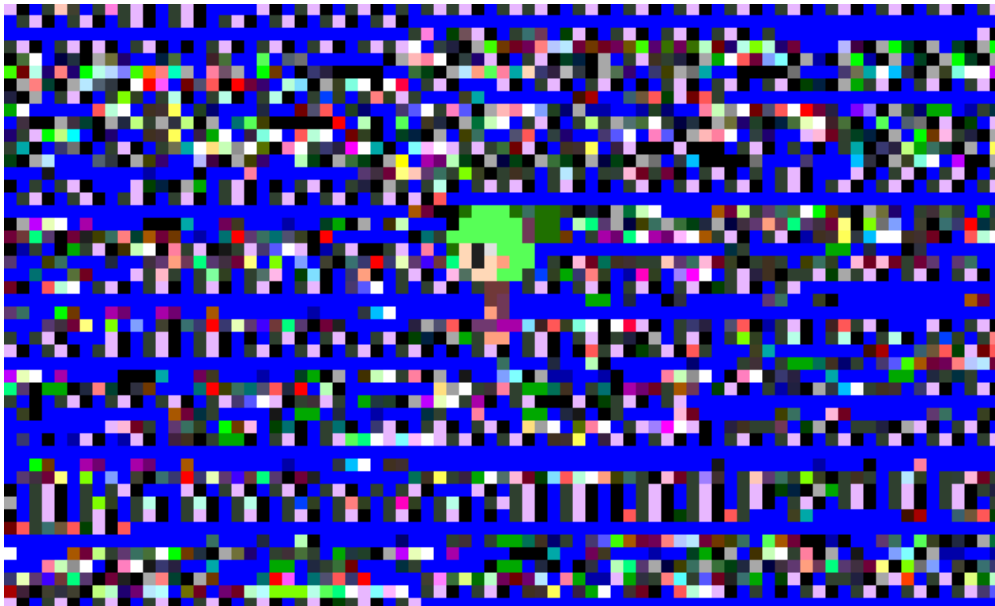
Also as I promised let’s end it with some (buggy) development screenshots:



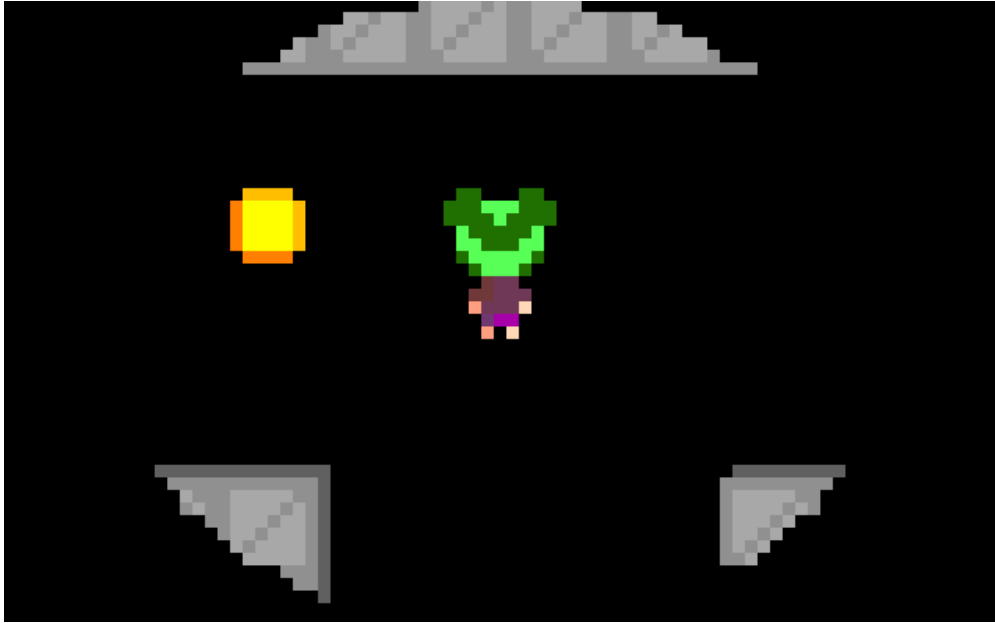
When writing the code scaling images up this happened



I messed up the address of the play sprite resulting in this



Here I loaded up the wrong address for the base of the map, resulting in rendering the code positioned there



When I optimized the world rendering I at some point choose a too small radius resulting in this

9 Sources and Resources

9.1 Part 1

9.2 Originally published at

- <https://0x00sec.org/t/realmode-assembly-writing-bootable-stuff-part-1/>
- <https://0x00sec.org/t/realmode-assembly-writing-bootable-stuff-part-2/>
- <https://0x00sec.org/t/realmode-assembly-writing-bootable-stuff-part-3/>
- <https://0x00sec.org/t/realmode-assembly-writing-bootable-stuff-part-4/>
- <https://0x00sec.org/t/realmode-assembly-writing-bootable-stuff-part-5/>
- <https://0x00sec.org/t/realmode-assembly-writing-bootable-stuff-part-6/>
- <https://0x00sec.org/t/realmode-assembly-writing-bootable-stuff-part-7/>

9.3 Part 1

9.3.1 Sources and References

- http://wiki.osdev.org/Real_Mode
- http://wiki.osdev.org/Protected_Mode
- https://en.wikipedia.org/wiki/Long_mode
- <https://en.wikipedia.org/wiki/BIOS>
- [http://wiki.osdev.org/MBR_\(x86\)](http://wiki.osdev.org/MBR_(x86))
- <http://wiki.osdev.org/Kernel>
- http://wiki.osdev.org/Boot_Sequence
- http://stanislavs.org/helppc/int_13.html
- https://en.wikipedia.org/wiki/INT_10H
- <http://duartes.org/gustavo/blog/post/how-computers-boot-up/>
- <http://flint.cs.yale.edu/feng/cos/resources/BIOS/>
- <http://gec.di.uminho.pt/discip/MaisAC/HCW/09R03.pdf>
- <https://superuser.com/questions/660143/does-the-bios-have-some-sort-of-generic-drivers>

9.4 Part 2

9.4.1 Link to source files

- <https://github.com/Pusty/realmode-assembly/tree/master/part2>

9.4.2 Other Hello World bootloaders (both don't load a kernel but are interesting anyways):

- http://viralpatel.net/taj/tutorial/hello_world_bootloader.php
- <http://blog.ackx.net/asm-hello-world-bootloader.html>

9.4.3 Sources:

- <http://www.nasm.us/>
- http://stanislavs.org/helppc/int_13-0.html
- <https://en.wikipedia.org/wiki/Cylinder-head-sector>
- <http://www.pcguide.com/ref/fdd/mediaGeometry-c.html>
- <http://mikeos.sourceforge.net/>
- http://stanislavs.org/helppc/int_13-2.html
- <http://www.delorie.com/djgpp/doc/rbinter/id/11/1.html>

9.5 Part 3

9.5.1 Link to source files

- <https://github.com/Pusty/realmode-assembly/tree/master/part3>

9.5.2 Sources and References

- <http://blog.ackx.net/asm-hello-world-bootloader.html>
- <http://wiki.osdev.org/QEMU>
- <http://mikeos.sourceforge.net/>
- https://en.wikipedia.org/wiki/BIOS_parameter_block
- <http://wiki.osdev.org/FAT>
- <http://f.osdev.org/viewtopic.php?f=1&t=28331>
- <https://stackoverflow.com/questions/1894843/how-can-i-put-a-compiled-boot-sector-onto-a-usb-stick>
- <https://www.cyberciti.biz/faq/howto-copy-mbr/>
- <https://bipedu.wordpress.com/2013/06/22/live-usb-linux-with-dd-command/>
- <https://askubuntu.com/questions/22381/how-to-format-a-usb-flash-drive>
- <http://hddguru.com/software/HDD-Raw-Copy-Tool/>
- <https://mh-nexus.de/en/hxd/>
- <https://www.windowscentral.com/how-clean-and-format-storage-drive-using-diskpart-windows-10>
- <http://www.qemu.org/>
- <http://www.nasm.us/>
- <http://www.mingw.org/>

9.6 Part 4

9.6.1 Link to source files

- <https://github.com/Pusty/realmode-assembly/tree/master/part4>

9.6.2 Sources and References

- http://stanislavs.org/helppc/int_16-0.html
- http://stanislavs.org/helppc/int_16-1.html
- http://stanislavs.org/helppc/scan_codes.html
- <https://0x00sec.org/t/counting-in-any-number-system/3254>
- <http://www.asciitable.com/>

9.7 Part 5

9.7.1 Link to source files

- <https://github.com/Pusty/realmode-assembly/tree/master/part5>

9.7.2 Sources and References

- https://en.wikipedia.org/wiki/Video_BIOS
- <https://www.pcorner.com/list/TUTOR/HELPPC21.ZIP/INTERRUP.TXT/>
- http://minuszerodegrees.net/video/bios_video_modes.htm
- https://en.wikipedia.org/wiki/Mode_13h
- https://en.wikipedia.org/wiki/BIOS_color_attributes
- <https://pdos.csail.mit.edu/6.828/2007/readings/hardware/vgadoc/VGABIOS.TXT>
- <http://www.brokenthorn.com/Resources/OSDevVga.html>
- https://upload.wikimedia.org/wikipedia/commons/6/66/VGA_palette_with_black_borders.svg

9.8 Part 6

9.8.1 Link to source files

- <https://github.com/Pusty/realmode-assembly/tree/master/part6>

9.8.2 Sources and References

- http://wiki.osdev.org/Drawing_In_Protected_Mode
- [http://wiki.osdev.org/Memory_Map_\(x86\)](http://wiki.osdev.org/Memory_Map_(x86))
- http://bos.asmhackers.net/docs/vga_without_bios/
- <http://wiki.osdev.org/Segmentation>
- <http://faydoc.tripod.com/cpu/stosb.htm>
- https://en.wikipedia.org/wiki/Multiple_buffering#Double_buffering_in_computer_graphics

9.9 Part 7

9.9.1 Link to source files

- <https://github.com/Pusty/realmode-assembly/tree/master/part7>

9.9.2 Sources and References

- https://wiki.osdev.org/Interrupt_Vector_Table
- <https://wiki.osdev.org/IDT>
- https://en.wikipedia.org/wiki/Interrupt_descriptor_table
- http://inglorion.net/documents/tutorials/x86ostut/keyboard/us_keymap.inc
- http://stanislavs.org/helppc/int_15-86.html