

Minimus OS

Luke

December 20, 2024

Abstract

This document serves as a reference for the Minimus operating system.

Contents

1	Running	4
1.1	Building	4
1.2	Emulation	4
2	Development	4
2.1	Kernel main	4
2.2	Using the crosscompiler	4
2.3	Including files	4

CONTINUES ON NEXT PAGE

3	Bootloader	5
3.1	Headers	5
3.2	Disk read	5
3.3	Usable memory	6
3.4	Display	7
3.4.1	VGA	7
3.4.2	Font	8
3.4.3	VBE	8
3.5	Enabling 32-bit processing	10
3.5.1	Segment descriptor	10
3.5.2	Prerequisites	11
3.5.3	Protected mode	11
3.6	Enabling the A20 line	12
3.7	Loading the Kernel	13
3.8	The boot signature	14
4	Kernel	15
4.1	Port Utils	15
4.2	Memory management	16
4.2.1	The heap	16
4.3	VGA/VBE	18
4.3.1	Initialisation	18
4.3.2	Setting pixel values	19
4.3.3	Drawing characters	20
4.4	Keyboard Input	21
4.4.1	Interrupt Descriptor Table	21
4.4.2	Program Interface Controller	22
4.5	File system	25
4.5.1	Reading from Disk	25
4.5.2	Creating a pagefile	27
4.5.3	Adding files to the file system	30
4.5.4	Executing files	33
5	Appendix	34
.1	INT 13 ₁₆ AH 02 ₁₆	34
.2	Segment Descriptor	34
.3	INT 15 ₁₆ EAX E820 ₁₆	34
.4	VGA Options	35
.5	VBE1 Functions	35
.6	INT 10 ₁₆ AX 4F02 ₁₆	36
.7	Enabling A20 Line through the Keyboard Controller	36

1 Running

1.1 Building

To build Minimus, a GNU Makefile has been included in the base directory, this can be run with the GNU Make tool, which is standard on most linux distros, and can be run with `make`.

On Windows, it is recommended to use WSL (Windows Subsystem for Linux) to build the program.

Running `make` also rebuilds the file system, clearing all files created by the Minimus kernel.

1.2 Emulation

To emulate minimus, I recommend either Bochs or Qemu. For bochs, the `bochsrc` file is included in the base directory (as `.bochsrc`), and can be started with `bochs -q`. For Qemu, you can emulate with `qemu-system-x86_64 -device ich9-intel-hda -hda bin/os.img`. You may need to use a VNC viewer to view the OS (at least I had to, as I was using wayland).

2 Development

2.1 Kernel main

The Minimus starting point is the `main()` function (found in `kernel/main.c`). It is integral to write your additional code after all of the `init...()` functions, as they are necessary for starting up Minimus and getting stuff going.

2.2 Using the crosscompiler

I included a compiler within this project in order to make applications for the Minimus operating system. This way, the Kernel `main()` function can stay quite small. The `crosscompiler/main.c` file will be used as the source file when `make` is run within the `crosscompiler` folder.

The makefile automatically links all Kernel functions, and more, in a sort of standard library to abstract away all the irritating low level functions, and the list of functions, their definitions, and usage is provided within the Standard Library section.

2.3 Including files

As well as the executable file, any other files referenced also have to be included within `file/files`, which will then be compiled into the Minimus file format (I haven't made a name for it yet). Executable files must be prefaced with 1, and other characters for file permissions have not yet been decided, for now, 0 will save for everything but executable files.

3 Bootloader

3.1 Headers

The disk is ordered into sectors, of size 200_{16} , starting from sector 1[3]. When the BIOS loads the OS, it copies the first sector, which would be the first 512 bytes, from the disk into memory, starting at $7C00_{16}$ [1].

Memory is also ordered into segments, of size 10_{16} [4]. This means that memory addresses can overlap, for example: The address $0000:7C00_{16}$ is the same as $0700:0C00_{16}$.

You must ensure that the bytes at $1FE_{16}$ and $1FF_{16}$ contain values 55_{16} and AA_{16} respectively[5]. This is called the magic number, and is used to differentiate bootable disks from non-bootable disks.

When the BIOS hands control to the OS, the CPU will be in 16 bit (real) mode[6], which means it is using 16 bits per instruction. This is because the CPU is in 16 bit mode by default. You must ensure that your program code for the bootloader begins in 16 bit (assuming real) mode.

bootloader/main.asm

```
1  [org 0x7c00]           ; memory load location
2  [bits 16]             ; real mode
```

3.2 Disk read

Reading from the disk is done with Bios Interrupt 13_{16} ah=02[2].

The interrupt specification is layed out below.1:

Register	Value
AH	02
AL	Number of sectors to read
CH	Cylinder
DH	Head
CL	Sector
DL	Drive
ES:BX	Output offset

Since the DL register is initialised with the boot drive before control is handed to our program[7], as long as it is not overwritten before calling int 13_{16} , you do not have to alter it.

Since the ES register cannot be written to directly, due to no CPU instruction being available to transfer a value to ES[8], which results in an intermediary value needing to be used.

I decided to use BX, due to it being overwritten used in the instruction after, but any unused register will do. Then, I assigned BX with $7e00_{16}$, which

is 200_{16} bytes more than the start of the program[1], which is the precise number of bytes loaded by the BIOS[3].

The number of load segments (register AX)[2] have a limit of around 100 on some systems, and around 70 on QEMU. 64 is a safe number of sectors to load, and allows for 32KB of kernel instructions (which is far more than enough).

Sometimes there will be an error, stored in the carry bit[2], if this happens, retry the load from disk. An error may also be indicated by the AL register having an incorrect number of sectors read.

bootloader/main.asm

```
1  KERNEL_SEGS equ 64          ; KERNEL_SEGS * 512
2
3  ; read kernel (https://en.wikipedia.org/wiki/INT\_13H)
4  mov bx, 0
5  mov es, bx
6  mov bx, 0x7e00              ; offset
7  mov ah, 0x02                ; set read mode
8  mov cl, 2                   ; start from sec 2
9  mov al, KERNEL_SEGS         ; sectors to read
10 mov ch, 0                   ; cylinder
11 mov dh, 0                   ; head
12 int 0x13                    ; call
13 jc $$                       ; carry bit stores error, loop
14 cmp al, KERNEL_SEGS         ; al is sectors read
15 jne $$                       ; if all sectors arent read, loop
```

3.3 Usable memory

The amount of usable memory varies between systems, and some memory is reserved for hardware, VGA and VBE, console video memory, etc.

Detecting available memory blocks is done with interrupt 15_{16} [12].3. Although you can probably get away with ignoring this, and just hoping memory above, lets say 1mb, is completely fine, I went through the extra effort to correctly get usable memory.

This area repeatedly calls int 15_{16} , and saves the result, if it is larger than the currently tested memory, in a memory location ironically not tested to see if it is reserved.

The copying and checking is done in two parts due to the fact that the numbers are larger than the 16 bit memory available in real mode.

bootloader/main.asm

```
1  ; get largest available memory block
2  pusha                      ; push all
3  mov cx, 0x0                ; clear cx for addition later
```

```

4  xor eax, eax
5  mov es, eax
6  mov ebx, 0x0                ; clear
7  mov edx, 0x534d4150         ; magic value
8  memreadloop:
9      mov eax, 0xe820          ; magic value
10     mov ecx, 0x18            ; magic value
11     mov di, 0x7bd0           ; memory location for buffer
12     int 0x15                ; call function
13     add di, cx               ; increment di by entry size
14                             ; (cx is 16 bit cl)
15 memreadloopvalid:
16     mov eax, [0x7be0]        ; load type
17     cmp eax, 1               ; check if 1 (available memory)
18     jne memreadloopend      ; go to next otherwise
19 memreadloopcheck:
20     mov eax, [0x7bda]        ; load size of current
21     mov ecx, [0x7bfa]        ; load size of biggest
22     cmp eax, ecx             ; check if bigger
23     jle memreadloopend      ; go to next otherwise
24 memreadlooprecord:
25     mov eax, [0x7bd0]        ; load address of current
26     mov [0x7bf0], eax        ; record biggest address
27     mov eax, [0x7bd4]        ; load address of current
28     mov [0x7bf4], eax        ; record biggest address
29     mov eax, [0x7bd8]        ; load size of current
30     mov [0x7bf8], eax        ; record biggest size
31     mov eax, [0x7bdc]        ; load size of current
32     mov [0x7bfc], eax        ; record biggest size
33 memreadloopend:
34     cmp ebx, 0               ; check if next
35     jnz memreadloop          ; repeat

```

3.4 Display

3.4.1 VGA

Enabling VGA[14] here allows me to get the font later, and also serves as compatibility for if VESA is not supported.

The options for AL are defined.4.

bootloader/main.asm

```

1  ; vga mode
2  mov ah, 0x0                ; graphics mode

```

```

3  mov al, 0x13                ; 256 colour 200x320
4  int 0x10                    ; set vga mode

```

3.4.2 Font

The code below gets the font from the VGA card. This is so I did not have to store my own.

The code is taken mostly from this OsDev page[13], due to the fact that I figured it was mostly copy paste anyway.

bootloader/main.asm

```

1  ; get vga font
2  mov eax, 0x110
3  mov es, eax
4  mov ax, 0x0
5  mov di, ax
6  push ds
7  push es
8  mov ax, 0x1130                ; magic numbers
9  mov bh, 0x6
10 int 0x10                    ; get vga font
11 ;mov ds, es
12 push es
13 pop ds
14 pop es
15 mov si, bp
16 mov cx, 0x400
17 rep movsd
18 pop ds

```

3.4.3 VBE

The VESA bios extensions allows for greater colour depth (RGB) and higher resolutions than VGA. Getting the VBE table is done with interrupt 10_{16} [15].

Once sifted to find the right function, it can again be enabled with interrupt 10_{16} , the magic numbers for each function are in the VBE spec[15].

The resulting values (whether 24 bit or 32 bit colour is used, the location of the framebuffer, whether VBE is even supported in the first place, etc), are placed into another (not checked) area of memory, which will be used by the kernel later.

I start by checking for VBE2 support, and if it doesn't exist I jump to the end of the VBE section. Then I go through each function, checking if it has RGB colourspace, and is the correct resolution. I then update the bytes at some position (again not fully checked) so my kernel has access to whether VBE was

enabled or not, and also to provide the kernel with the position of the linear framebuffer.

This is due to the fact that in VBE2, it is not required that all the VBE1 functions.⁵ are supported (even though they most likely are).

Enabling a VBE mode is as simple as VBE once the functions have been sorted, and is the same BIOS interrupt.⁶

bootloader/main.asm

```
1  ; get vesa support
2  mov [0x2000], DWORD "VBE2"
3  mov ax, 0x4f00                ; magic number
4  mov di, 0
5  mov es, di
6  mov di, 0x2000                ; offset to table
7  int 0x10                      ; get vbe table
8  cmp ax, 0x004f                ; check if support
9  jne skip_vbe                  ; use vga instead if not
10
11 mov di, 0                      ; offset to tmp
12 mov es, di
13 mov di, 0x2200                ; segment to tmp
14 mov si, [0x2000 + 16]         ; segment of list
15 mov ds, si
16 mov si, [0x2000 + 14]         ; offset of list
17
18 vbe_loop:
19 ; get next supported vesa function
20 movsw
21 sub di, 2                      ; (2 is added to both)
22 mov cx, [0x2200]              ; vesa mode
23 cmp cx, 0xffff                ; check if end
24 je skip_vbe                   ; loop
25
26 ; get details
27 mov ax, 0x4f01                ; magic number
28 int 0x10                      ; get vbe
29
30 mov ax, [0x2200]              ; check attr
31 and ax, 0b10000000            ; check bit 7 (linear framebuffer)
32 cmp ax, 0
33 jz vbe_loop                   ; if not linear, loop
34
35 mov al, [0x2200 + 25]          ; check colourspace
36 mov [0x2201], BYTE 0x00        ; vbe 32 bit flag
37 cmp al, 24
```

```

38 je vbe_check_dims          ; if not rgb, loop
39 mov [0x2201], BYTE 0xff    ; vbe 32 bit flag
40 cmp al, 32
41 je vbe_check_dims          ; if not rgb, loop
42 jmp vbe_loop                ; actual loop
43
44 vbe_check_dims:
45 mov ax, [0x2200 + 18]       ; check width
46 cmp ax, 640
47 jne vbe_loop                ; if not desired, loop
48
49 mov ax, [0x2200 + 20]       ; check height
50 cmp ax, 480
51 jne vbe_loop                ; if not desired, loop
52
53 jmp enable_vbe              ; turn on this vbe
54
55 skip_vbe:
56 mov [0x2200], BYTE 0x00     ; vbe correct flag
57 jmp complete_vbe            ; skip
58
59 enable_vbe:
60 mov ax, [0x2200 + 40]       ; framebuffer
61 mov [0x2200 + 2], ax        ; framebuffer
62 mov ax, [0x2200 + 42]       ; framebuffer
63 mov [0x2200 + 4], ax        ; framebuffer
64 mov ax, 0x4f02              ; magic number
65 mov bx, cx                  ; move mode number
66 or bx, 0x4000               ; set linear framebuffer
67 int 0x10                    ; set vbe mode
68 cmp ax, 0x004f              ; test for error
69 jne skip_vbe                ; skip if error (will use vga
    ↪ instead)
70 mov [0x2200], BYTE 0xff     ; vbe correct flag
71
72 complete_vbe:
73 popa                        ; pop all

```

3.5 Enabling 32-bit processing

3.5.1 Segment descriptor

The Global Descriptor Table is necessary for enabling protected mode[9], and is defined below in a basic form in order to switch to protected mode.

For my kernel, none of the features of the GDT, like paging[11], are not needed, so the GDT is very basic.2.

bootloader/main.asm

```
1  jmp gdt_after
2
3  ; segment descriptor (reverse order)
4  gdt_start:
5      dq 0                ; null byte start
6  gdt_code:
7      dw 0xffff           ; segment limit
8      db 0,0,0           ; segment base
9      db 0b10011010       ; flags (see wiki)
10     db 0b11001111       ; 4b flags (see wiki) + seg limit
11     db 0                ; segment base
12 gdt_data:
13     dw 0xffff           ; segment limit
14     db 0,0,0           ; segment base
15     db 0b10010010       ; flags (see wiki)
16     db 0b11001111       ; 4b flags (see wiki) + seg limit
17     db 0                ; segment base
18 gdt_end:
19     dw gdt_end - gdt_start - 1 ; limit
20     dd gdt_start         ; addr 24 bit
21 gdt_after:
```

3.5.2 Prerequisites

To enable protected mode, you must first disable interrupts, and load the global descriptor table[16], which can be done in NASM with commands `cli` and `lgdt [GDT ADDRESS]` respectively.

bootloader/main.asm

```
1  cli                ; disable interrupts
2  lgdt [gdt_end]     ; gdt_end is descriptor table
   ↪ descriptor
```

3.5.3 Protected mode

Upon the completion of all the above, you can then set bit 0 (starting from 0) to 1 in the control register CR0[16], which must be done in separate commands, as the special register CR0 is not directly assignable[8].

You must then immediately[16] perform a long jump to your next desired instruction.

You then have to tell the assembler that you are now using 32 bits, which can be done with the `[bits 32]` command in NASM.

bootloader/main.asm

```
1  mov eax, cr0
2  or  eax, 1          ; set 1 bit in control register for
   ↪ protected mode
3  mov cr0, eax
4  jmp (gdt_code - gdt_start):bits32code ; stall cpu and
   ↪ flush all cache (as moving to different segment) to finalize
   ↪ protected mode
5
6  ; finally 32 bits
7  [bits 32]
8  bits32code:
```

3.6 Enabling the A20 line

Enabling the A20 line (the proper way) is by the keyboard[10]. This is incredibly tedious and repetitive, which is why I have taken this code from the OsDev page.

Two small assembly functions: a20wait; a20waitr, have been created to avoid some of the repetition. a20wait will repeatedly poll port 64₁₆ until bit 1 is set (which is why TEST is used). a20waitr will do the same until bit 0 is set.

This is due to the fact that it is required to wait for bit 1 in port 64₁₆ before writing and bit 0 in port 64₁₆ before reading.

A brief outline of the functions of the A20 line are provided within this document.7.

bootloader/main.asm

```
1  xor al, al
2  call a20wait          ; wait for write
3  mov al, 0xad
4  out 0x64, al          ; send 0xad
5  call a20wait          ; wait for write
6  mov al, 0xd0
7  out 0x64, al          ; send 0xd0
8  call a20waitr         ; wait for read
9  in  al, 0x60           ; get ack
10 push eax
11 call a20wait          ; wait for write
12 mov al, 0xd1
13 out 0x64, al          ; send 0xd1
14 call a20wait          ; wait for write
15 pop eax              ; eax gets overwritten
16 or  al, 0b0010        ; a20 bit
17 out 0x60, al          ; set a20 bit on
18 call a20wait          ; wait for write
```

```

19 mov al, 0xae
20 out 0x64, al                ; send 0xae
21 call a20wait                ; wait for generic
22 jmp skipa20                 ; go to end
23 a20wait:
24     in al, 0x64
25     test al, 2
26     jnz a20wait
27     ret
28 a20waitr:
29     in al, 0x64
30     test al, 1
31     jz a20waitr
32     ret
33
34 skipa20:

```

3.7 Loading the Kernel

Before loading the kernel, I initialise all the registers high parts with the segment and move the stack pointers to 7000_{16} .

The bootloader begins at $7c00_{16}[1]$, and because the stack grows downwards, 7000_{16} is a perfect location for a stack. You can place yours where you like.

bootloader/main.asm

```

1 jmp (gdt_code - gdt_start):start_kernel
2
3 start_kernel:
4     ; segment registers init
5     mov ax, gdt_data - gdt_start
6     mov ds, ax
7     mov es, ax
8     mov fs, ax
9     mov gs, ax
10    mov ss, ax
11
12    ; stack pointers
13    mov esp, 0x7000                ; top of stack
14    mov ebp, esp                  ; bottom of stack
15
16    call kernel                    ; start kernel and move back
17    ↵ to segment
18
19 ; kernel
20 kernel:

```

```

20         jmp kernel_loadseg      ; hand control to kernel
21         jmp $$                  ; return -> error, loop
22
23         ...
24
25     ; kernel load
26 kernel_loadseg:
27     call kernel_cseg
28     jmp $                        ; if fail restart
29 kernel_cseg:                    ; compiled c appeneded here

```

3.8 The boot signature

You must ensure that the bytes at $1FE_{16}$ and $1FF_{16}$ contain values 55_{16} and AA_{16} respectively[5]. This is called the boot signature, or magic number, and is used to differentiate bootable disks from non-bootable disks, and also tells the BIOS how to load your OS.

bootloader/main.asm

```

1     ; padding
2     times 510 - ($-$$) db 0
3
4     ; boot signature
5     db 0x55,0xaa

```

4 Kernel

The kernel is mostly written in C, with occasional exceptions like the IDT, which is written in NASM assembly.

4.1 Port Utils

Many kernel functions require reading and writing data to the serial bus, so I have made functions for code readability.

GCC inline assembly has three major parts: the instruction - which is written in GNU assembly; the input value - (e.g. "=a"(val) to save to val); the output value - (e.g. "a"(val) to load from val), all separated by colons.

kernel/libs/ioutils.c

```
1 void outb(unsigned short port, unsigned char val) {
2     __asm__ volatile ("outb %b0, %w1" : : "a"(val),
3         ↪ "Nd"(port) : "memory");
4 }
5
6 unsigned char inb(unsigned short port) {
7     unsigned char _ret = 0;
8
9     __asm__ volatile ("inb %w1, %b0" : "=a"(_ret) :
10         ↪ "Nd"(port) : "memory");
11
12     return _ret;
13 }
14
15 void outw(unsigned short port, unsigned short val) {
16     __asm__ volatile ("outw %w0, %w1" : : "a"(val),
17         ↪ "Nd"(port) : "memory");
18 }
19
20 unsigned short inw(unsigned short port) {
21     unsigned short _ret = 0;
22
23     __asm__ volatile ("inw %w1, %w0" : "=a"(_ret) :
24         ↪ "Nd"(port) : "memory");
25
26     return _ret;
27 }
28
29 void iowait() {
30     outb(0x80, 0);
31 }
```

4.2 Memory management

4.2.1 The heap

The two most common ways to manage memory are the stack and the heap. We already have a stack, which is defined as 7000_{16} , and grows downwards, but we need a way to access more, and larger, memory, at random.

I had already wrote a heap before, but had used the concept of object orientation, due to the fact I wrote it in my (then and still now) favourite language C++, which I cannot stop glazing (I believe my opinion may change after I try Rust).

The heap is a large area of semi-organised data. The way I created my heap, it is organised into blocks, and each block points to both the previous and next block. As well as this, the size, and whether the block is occupied is stored.

When giving memory, I simply add the size of `memchunk` onto the pointer. I decided that no space for overflowing values will be required for this kernel.

The main functions, excluding the utility functions and those required by the GCC compiler, are defined below, including how I implemented my heap.

kernel/libs/vga.c

```
1  #include "ioutils.h"
2
3  struct memchunk {
4      struct memchunk* prev;
5      struct memchunk* next;
6      unsigned long size;
7      char occupied;
8  };
9
10 struct memchunk* heap;
11 void initheap() {
12     heap = (struct memchunk*)(*(unsigned long*)0x7bf0);
13     heap->size = *(unsigned long*)0x7bf8;
14     heap->occupied = 0;
15 }
16
17 void* malloc(unsigned int size) {
18     struct memchunk* check = heap;
19     do {
20         if (check->size > size + 16 && !check->occupied)
21             ↪ {
22                 struct memchunk* nextchunk = (struct
23                 ↪ memchunk*)(check + size + 16);
24
25                 nextchunk->size = check->size - size -
26                 ↪ 16;
```



```

24         nextchunk->prev = check;
25         nextchunk->next = check->next;
26         nextchunk->occupied = 0;
27
28         check->next = nextchunk;
29         check->size = size;
30         check->occupied = 1;
31
32         return check + 16;
33     }
34
35     check = check->next;
36 } while (check);
37 return 0;
38 }
39
40 struct memchunk* m_memchunkstartfreesegment(struct memchunk*
↪ check) {
41     if (check->prev && !check->prev->occupied)
42         return m_memchunkstartfreesegment(check->prev);
43     return check;
44 }
45
46 int free(void* ptr) {
47     struct memchunk* check = ptr - 16;
48
49     if ((check->prev && check->prev->next != check) ||
↪ (check->next && check->next->prev != check)) {
50         return 1;
51     }
52
53     check->occupied = 0;
54
55     check = m_memchunkstartfreesegment(check);
56     while (check->next && !check->next->occupied) {
57         check->size += check->next->size;
58         check->next = check->next->next;
59     }
60     if (check->next)
61         check->next->prev = check;
62
63     return 0;
64 }
65
66 void* realloc(void* ptr, unsigned int size) {
67     struct memchunk* check = ptr - 16;

```

```

68     if (check->next && check->next->size >= size -
        ↪ check->size) { // 16 bit excluded as it is on both
        ↪ sides
69         check->size += check->next->size;
70         check->next = check->next->next;
71         return ptr;
72     }
73
74     // slower solution if next memory block isnt free
75     void* _ptr = malloc(size);
76
77     for (unsigned int i = 0; i < size; i++) {
78         *((char*)_ptr+i) = *((char*)ptr+i);
79     }
80
81     free(ptr);
82
83     return _ptr;
84 }

```

4.3 VGA/VBE

VGA is outdated, so is only used as a backup in this OS, in case VBE2 fails to load for whatever reason, or is not supported.

4.3.1 Initialisation

VGA memory is set at $A0000_{16}$, whereas VBE uses a variable position linear framebuffer. In the bootloader, when VBE2 was enabled, the linear framebuffer was stored at 2200_{16} , so it is loaded from there.

Additionally, whether VBE was successfully enabled is loaded, as well as how many bits per pixel are used (on some systems it is 4).

The width and height is pre-defined for this kernel, so is not needed to be gathered from anywhere.

For compatibility, the VGA colourspace is loaded with a compressed version of RGB (for 8 bits: RRRGGGBB).

kernel/libs/vga.c

```

1  #include "ioutils.h"
2
3  extern void getvgafont();
4
5  #define VGA_MEM (unsigned char*)0xa0000
6  #define VGA_WIDTH 320
7  #define VGA_HEIGHT 200

```

```

8
9  #define VBE_WIDTH 640
10 #define VBE_HEIGHT 480
11 unsigned char vbeEnabled = 0;
12 unsigned char vbeAlpha = 3;
13 unsigned char* vbeFramebuffer = (unsigned char*)0x0;
14
15 unsigned char* font = (unsigned char*)0x1100; // defined in
    ↪ bootloader/main.asm
16
17 void initvga() {
18     vbeEnabled = *(unsigned char*)0x2200;
19     if (*(unsigned char*)0x2201)
20         vbeAlpha = 4;
21     else
22         vbeAlpha = 3;
23
24     if (vbeEnabled) {
25         vbeFramebuffer = (unsigned char*)(*(unsigned
    ↪ int*)0x2202);
26     }
27     else {
28         for (int i = 0; i < 256; i++) {
29             unsigned char r = (i & 0b11100000) >> 2;
    ↪ // highest value is 0b00111111 as 18
    ↪ bit colour space
30             unsigned char g = (i & 0b00011100) << 1;
31             unsigned char b = (i & 0b00000011) << 4;
    ↪ // used less bits for blue as eyes
    ↪ are less sensitive
32
33             outb(0x3c8, i); // set DAC address
34             outb(0x3c9, r); // set DAC R for i
35             outb(0x3c9, g); // set DAC G for i
36             outb(0x3c9, b); // set DAC B for i
37         }
38     }
39 }

```

4.3.2 Setting pixel values

Setting pixel values in VGA and VBE are similar, with the exception that in VBE, you may need to have an extra alpha bit that is left alone. I handled this case as below.

In VGA, I have compressed the colourspace down to the values allowed, and

while this looks low quality, it serves as a great fallback for old systems with very little overhead.

kernel/libs/vga.c

```
1 void drawpixel(int _x, int _y, unsigned char r, unsigned char g,  
  ↪ unsigned char b) {  
2     if (vbeEnabled) {  
3         *(vbeFramebuffer + _x * vbeAlpha + vbeAlpha - 3  
  ↪ + _y * VBE_WIDTH * vbeAlpha) = b;  
4         *(vbeFramebuffer + _x * vbeAlpha + vbeAlpha - 2  
  ↪ + _y * VBE_WIDTH * vbeAlpha) = g;  
5         *(vbeFramebuffer + _x * vbeAlpha + vbeAlpha - 1  
  ↪ + _y * VBE_WIDTH * vbeAlpha) = r;  
6     }  
7     else {  
8         _x /= 2;  
9         _y /= 3;  
10        *(VGA_MEM + _x + _y * VGA_WIDTH) = (r &  
  ↪ 0b11100000) | ((g >> 3) & 0b00011100) | ((b  
  ↪ >> 6) & 0b00000011);  
11    }  
12 }
```

4.3.3 Drawing characters

For drawing characters, you need a font. Although the kernel currently has a filesystem, at the time I implemented the display, it did not.

I decided to append the bootloader's VGA code with code to load the VGA BIOS currently used font (as it was easier to do this with BIOS interrupts).

This code is covered in the Bootloader chapter, so I will not be covering this. The pointer has been assigned to the `font` variable.

kernel/libs/vga.c

```
1 void drawchar(int _x, int _y, unsigned char _char) {  
2     unsigned char* fontchar = font + ((int)_char * 16);  
3  
4     for (int y = 0; y < 16; y++) {  
5         for (int x = 0; x < 8; x++) {  
6             if (fontchar[y] & (0b10000000 >> x)) {  
7                 drawpixel(_x * 8 + x, _y * 16 +  
  ↪ y, 0xff, 0xff, 0xff);  
8             }  
9             else {  
10                drawpixel(_x * 8 + x, _y * 16 +  
  ↪ y, 0x00, 0x00, 0x00);  
11            }  
12        }  
13    }  
14 }
```

```

11         }
12     }
13 }
14 }

```

4.4 Keyboard Input

Getting input from the keyboard, mouse, clock, and others is done through the Program Interface Controller, which is a way of sending immediate jobs to the CPU, called interrupts.

4.4.1 Interrupt Descriptor Table

The Interrupt Descriptor Table[17] is an array of values, where each index corresponds to an interrupt, and each value the memory location of that interrupt function.

kernel/libs/interrupts.c

```

1  global initidtasm
2  global interrupthandler
3  global idtdescriptor
4  global idtaddr
5
6  section .data
7
8  idtdescriptor:
9      dw 256*8-1          ; size
10 idtaddr:
11     dd 0x10000          ; address
12
13 section .text
14
15 initidtasm:
16     lidt [idtdescriptor]
17     ret
18
19 %macro idt 1
20     extern idt%1
21     global _idt%1
22     _idt%1:
23         pusha
24         call idt%1
25         popa
26         iret
27 %endmacro

```

```

28
29 idt ...

```

kernel/libs/interrupts.c

```

1  #include "ioutils.h"
2
3  void interrupthandler(unsigned char interrupt);
4
5  struct idtelement {
6      unsigned short offset1; // offset 0-15
7      unsigned short selector; // a code segment selector in
8      ↪ gdt
9      unsigned char base; // segment base (reserved) + ist
10     unsigned char attr; // gate type, dpl, and leftmost bit
11     ↪ must be 1
12     unsigned short offset2; // offset 16-31
13 };
14
15 extern void* idtaddr;
16
17 // https://en.wikipedia.org/wiki/Interrupt_descriptor_table
18 void initidtelement(unsigned int num, unsigned int func,
19 ↪ unsigned char trap) {
20     struct idtelement* element = (struct
21     ↪ idtelement*)(idtaddr+num*8);
22     element->base = 0;
23     element->selector = 8;
24     element->attr = 0b10001110 | trap;
25     element->offset1 = (unsigned short)(func);
26     element->offset2 = (unsigned short)(func >> 16);
27 }

```

4.4.2 Program Interface Controller

The PIC is what sends the signals for the interrupt to the CPU, so once interrupts have been tested and are working, the PIC must be programmed.

The PIC should be set, to begin with, so that all interrupts (except the communication bus) are masked, and are unmasked as you need them. This is to save on processor time.

You must first disable interrupts, then send the ICW1 signal to both the slave and master PIC (port 20₁₆ and A0₁₆ respectively). Then, you must set the vector offsets (ICW2), inform the PIC chips of each others existence (ICW3), and finally, enable 8086 mode (ICW4)[17].

You are then able to set the mask such that only the chips can communicate, and enable the required PIC interrupts as you go along in your kernel.

kernel/libs/interrupts.c

```
1  #define PIC1 0x20 // master pic
2  #define PIC2 0xa0 // slave pic
3
4  #define PIC1_OFFSET 0x20
5  #define PIC2_OFFSET (PIC1_OFFSET+8)
6
7  void sendeoi(unsigned char reg) {
8      if (reg >= 8) {
9          outb(PIC2, 0x20);
10     }
11     outb(PIC1, 0x20);
12 }
13
14 // https://wiki.osdev.org/8259_PIC#Programming_the_PIC_chips
15 void initpic() {
16     // disable interrupts
17     __asm__ volatile ("cli");
18
19     // each wait allows PIC to process
20
21     // set initialisation command for cascade (master and
22     ↪ slave) (icw1)
23     outb(PIC1, 0x11);
24     iowait();
25     outb(PIC2, 0x11);
26     iowait();
27
28     // set vector offsets (icw2)
29     outb(PIC1+1, PIC1_OFFSET);
30     iowait();
31     outb(PIC2+1, PIC2_OFFSET);
32     iowait();
33
34     // inform master of slave pic (icw3)
35     outb(PIC1+1, 0b0100);
36     iowait();
37     // inform slave of slave pic (icw3)
38     outb(PIC2+1, 0b0010);
39     iowait();
40
41     // enable 8086 mode (icw4)
42     outb(PIC1+1, 0b0001);
43     iowait();
44     outb(PIC2+1, 0b0001);
```

```

44         iowait();
45
46         // masks
47         outb(PIC1+1, 0xff & ~(1 << 2));
48         outb(PIC2+1, 0xff);
49
50         // enable interrupts
51         __asm__ volatile ("sti");
52     }
53
54     void enablepic(unsigned char irq) {
55         unsigned short port = PIC1+1;
56
57         if (irq >= 8) {
58             port = PIC2+1;
59             irq -= 8;
60         }
61
62         unsigned char val = inb(port) & ~(1 << irq); // remove
63         ↪ mask bit (set 0)
64         outb(port, val);
65     }
66
67     void disablepic(unsigned char irq) {
68         unsigned short port = PIC1+1;
69
70         if (irq >= 8) {
71             port = PIC2+1;
72             irq -= 8;
73         }
74
75         unsigned char val = inb(port) | (1 << irq); // add mask
76         ↪ bit (set 1)
77         outb(port, val);
78     }
79
80     //
81     ↪ https://pdos.csail.mit.edu/6.828/2014/readings/hardware/8259A.pdf
82     // 4. INTERRUPT REQUEST REGISTER (IRR) AND IN-SERVICE REGISTER
83     ↪ (ISR)
84     unsigned short irqreg(int cmd) {
85         outb(PIC1, cmd);
86         outb(PIC2, cmd);
87         return (inb(PIC2) << 8) | inb(PIC1);
88     }

```



```

86 #define idtbody(x, y) \
87 extern unsigned int _idt##y; \
88 void idt##y() { \
89     interrupthandler(x); \
90     sendeoi(x); \
91 } \
92
93 #define idthead(x, y) \
94 initidtelement(y, (unsigned int)&_idt##y, 0); \
95 enablepic(x); \

```

A helper function for the PIC is also defined, allowing easy binding of multiple functions to an interrupt.

kernel/libs/pic.c

```

1 void (*picFunc[16][16])();
2
3 int addPicFunc(int pic, void (*func)()) {
4     for (int i = 0; i < 16; i++) {
5         if (!picFunc[pic][i]) {
6             picFunc[pic][i] = func;
7             return 0;
8         }
9     }
10    return -1;
11 }
12
13 // interrupts.h
14 void interrupthandler(unsigned char interrupt) {
15     for (int i = 0; i < 16; i++)
16         if (picFunc[interrupt][i])
17             picFunc[interrupt][i]();
18 }

```

4.5 File system

4.5.1 Reading from Disk

To read from the disk, you can switch back to real mode, or you can create a device driver.

I decided to create a simple ATA driver, as it would work with most types of disks (due to most being compatible with ATA).

There are many guides on ATA controllers, so I will not go into the details within this document. You could even use somebody else's, but I believe that the educational benefit of creating a device driver

The main principle of reading from the disk with ATA is that it is read in segments, and each segment is read at a certain time. You can figure out that time by either polling, or interrupts. I decided on polling as it is faster for now, but I may change the design later on.

kernel/libs/disk.c

```

1  #include "ioutils.h"
2  #include "memory.h"
3
4  void diskWait() {
5      for (int i = 0; i < 15; i++) {
6          inb(0x1f7);
7      }
8      unsigned char status = inb(0x1f7);
9      while ((status & 0b10000000 && !(status & 0b1000)))
10         status = inb(0x1f7);
11 }
12
13 void* diskReadSector(unsigned int lba, unsigned char sectors) {
14     unsigned short mallocSec = sectors;
15     if (mallocSec == 0)
16         mallocSec = 256;
17     unsigned char* buffer = malloc(mallocSec * 512);
18
19     outb(0x1f6, 0xe0 | ((lba >> 24) & 0x0f)); // drive and
20     ↪ upper 4 bits of lba
21     outb(0x1f1, 0); // ignored but necessary on some systems
22     outb(0x1f2, sectors);
23     outb(0x1f3, lba); // lower 8 bits
24     outb(0x1f4, lba >> 8); // mid lower 8 bits
25     outb(0x1f5, lba >> 16); // mid upper 8 bits
26     outb(0x1f7, 0x20); // read flag
27
28     for (int sector = 0; sector < mallocSec; sector++) {
29         diskWait();
30
31         for (int i = 0; i < 256; i++) {
32             *(unsigned short*)(buffer + sector * 512
33             ↪ + i * 2) = inw(0x1f0);
34         }
35     }
36
37     return buffer;
38 }

```

```

38 void diskWriteSector(unsigned int lba, unsigned char sectors,
    ↪ unsigned char* buffer) {
39     unsigned short mallocSec = sectors;
40     if (mallocSec == 0)
41         mallocSec = 256;
42
43     outb(0x1f6, 0xe0 | ((lba >> 24) & 0x0f)); // drive and
    ↪ upper 4 bits of lba
44     outb(0x1f1, 0); // ignored but necessary on some systems
45     outb(0x1f2, sectors);
46     outb(0x1f3, lba); // lower 8 bits
47     outb(0x1f4, lba >> 8); // mid lower 8 bits
48     outb(0x1f5, lba >> 16); // mid upper 8 bits
49     outb(0x1f7, 0x30); // write flag
50
51     for (int sector = 0; sector < mallocSec; sector++) {
52         diskWait();
53
54         for (int i = 0; i < 256; i++) {
55             outw(0x1f0, *(unsigned int*)(buffer +
    ↪ sector * 512 + i * 2));
56         }
57     }
58 }

```

4.5.2 Creating a pagefile

A pagefile is just a giant list of filenames and where that filename points to on the disk. It is similar to the implementation of the heap, except that all the list begins at the start, and is in one place. This saves us from having to check many different sectors on our disk for small pieces of fragmented information.

For this kernel, since the amount of files will be relatively low, I have decided to load all of the files into memory. This, although inefficient on memory, will have hardly any impact. For example, a thousand files (with reasonable filename sizes of 10 characters) will take up only 19kB, which is far less memory than is needed.

It is for this reason that I have decided to load all of the filepage into memory, as the cost of loading them in and out of memory is (probably) more than the kilobytes of memory you will save.

On user based operating systems, this cost isnt negligible, as they may have thousands of files, but I can easily change this code later on, so I have decided to do it this way.

kernel/libs/file.c

```

1  #include "disk.h"

```

```

2  #include "strutils.h"
3  #include "memory.h"
4
5  #define PAGEADDRDISK 0x5000
6
7  struct filePage {
8      unsigned long address;
9      unsigned long size;
10     char name;
11 };
12 #define filePageSize 20
13
14 struct filePage* pageaddr;
15
16 struct filePage* fileDescriptor(char* filename) {
17     struct filePage* ptr = pageaddr;
18
19     while (ptr->address) {
20         if (!strcmp(&(ptr->name), filename)) {
21             return ptr;
22         }
23         *((unsigned char*)&ptr) += strlen(&(ptr->name))
24         ↪ + filePageSize;
25     }
26
27     return (struct filePage*)0;
28 }
29
30 void* fileRead(char* filename) {
31     struct filePage* descriptor = fileDescriptor(filename);
32
33     if (descriptor)
34         return diskRead(descriptor->address,
35         ↪ descriptor->size);
36     else
37         return (void*)0;
38 }
39
40 void fileDelete(char* filename) {
41     struct filePage* descriptor = fileDescriptor(filename);
42     if (!descriptor)
43         return;
44     unsigned int len = strlen(&(descriptor->name)) +
45     ↪ filePageSize;
46     unsigned char* ptr = (unsigned char*)descriptor;

```

```

45     unsigned int conseqZero = 0;
46     for (unsigned long i = 0; conseqZero < len; i++) {
47         ptr[i] = ptr[i + len];
48         if (ptr[i] == 0)
49             conseqZero++;
50         else
51             conseqZero = 0;
52     }
53 }
54
55 void fileWrite(char* filename, unsigned char* buffer, unsigned
↪ long size) {
56     fileDelete(filename);
57
58     struct filePage* descriptor = pageaddr;
59     unsigned long descaddr = 0;
60
61     while (descriptor->address) {
62         descaddr = descriptor->address +
↪ descriptor->size;
63         *((unsigned char*)&descriptor) +=
↪ strlen(&(descriptor->name)) + filePageSize;
64     }
65
66     descriptor->address = descaddr;
67     descriptor->size = size;
68     memcpy(&(descriptor->name), filename, strlen(filename));
69
70     diskWriteSector(PAGEADDRDISK / 512, 0, (void*)pageaddr);
71
72     diskWrite(descriptor->address, size, buffer);
73 }
74
75 char** fileList() {
76     int sizeTrack = sizeof(void*);
77     char** wordList = malloc(sizeTrack);
78     wordList[0] = (char*)0;
79
80     struct filePage* ptr = pageaddr;
81
82     while (ptr->address) {
83         sizeTrack += sizeof(void*);
84         wordList = realloc(wordList, sizeTrack);
85         wordList[sizeTrack / sizeof(void*) - 2] =
↪ &(ptr->name);
86         wordList[sizeTrack / sizeof(void*) - 1] = 0;

```

```

87         *((unsigned char*)&ptr) += strlen(&(ptr->name))
           ↪ + filePageSize;
88     }
89
90     return wordList;
91 }
92
93 void initfs() {
94     pageaddr = (struct filePage*)diskReadSector(PAGEADDRDISK
           ↪ / 512, 0);
95     pageaddr->address = PAGEADDRDISK + 256 * 512;
96     pageaddr->size = 0;
97     pageaddr->name = 0;
98
99     diskWriteSector(PAGEADDRDISK / 512, 1, (void*)pageaddr);
100 }

```

4.5.3 Adding files to the file system

To put files on the operating system, maybe ones you need by default, I have added a small file compiler in the files folder.

files/compiler.c

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include <dirent.h>
5
6  #define BUFSIZE 0x1000000
7  #define FILEDIR "files"
8  #define PAGEADDRDISK 0x5000
9
10 struct filePage {
11     unsigned int address;
12     unsigned int size;
13     char name;
14 };
15 #define filePageSize 20
16
17 struct filePage* pageaddr;
18
19 struct filePage* fileDescriptor(char* filename) {
20     struct filePage* ptr = pageaddr;
21
22     while (ptr->address) {

```

```

23         if (!strcmp(&(ptr->name), filename)) {
24             return ptr;
25         }
26         *((unsigned char*)&ptr) += strlen(&(ptr->name))
        ↪ + filePageSize;
27     }
28
29     return (struct filePage*)0;
30 }
31
32 void fileDelete(char* filename) {
33     struct filePage* descriptor = fileDescriptor(filename);
34     if (!descriptor)
35         return;
36     unsigned int len = strlen(&(descriptor->name)) +
        ↪ filePageSize;
37     unsigned char* ptr = (unsigned char*)descriptor;
38
39     unsigned int conseqZero = 0;
40     for (unsigned long i = 0; conseqZero < len; i++) {
41         ptr[i] = ptr[i + len];
42         if (ptr[i] == 0)
43             conseqZero++;
44         else
45             conseqZero = 0;
46     }
47 }
48
49 void fileWrite(char* filename, FILE* fileptr, unsigned int size)
        ↪ {
50     fileDelete(filename);
51
52     struct filePage* descriptor = pageaddr;
53     unsigned long descaddr = 0;
54
55     while (descriptor->address) {
56         descaddr = descriptor->address +
        ↪ descriptor->size;
57         *((unsigned char*)&descriptor) +=
        ↪ strlen(&(descriptor->name)) + filePageSize;
58     }
59
60     descriptor->address = descaddr;
61     descriptor->size = size;
62     memcpy(&(descriptor->name), filename, strlen(filename));
63

```

```

64         fread((unsigned char*)pageaddr + descriptor->address -
65             ↪ PAGEADDRDISK, size, 1, fileptr);
66     }
67     void main() {
68         pageaddr = malloc(BUFSIZE);
69         for (unsigned long i = 0; i < BUFSIZE; i++) {
70             ((unsigned char*)pageaddr)[i] = 0;
71         }
72         pageaddr->address = PAGEADDRDISK + 256 * 512;
73         pageaddr->size = 0;
74         pageaddr->name = 0;
75
76         DIR* d;
77         struct dirent* dir;
78         d = opendir(FILEDIR);
79         if (d) {
80             while ((dir = readdir(d)) != NULL) {
81                 if (!strcmp(dir->d_name, ".") ||
82                     ↪ !strcmp(dir->d_name, ".."))
83                     continue;
84                 printf("Compiling %s\n", dir->d_name);
85
86                 char path[strlen(dir->d_name) + 7];
87                 for (int i = 0; i < strlen(FILEDIR);
88                     ↪ i++) {
89                     path[i] = FILEDIR[i];
90                 }
91                 path[strlen(FILEDIR)] = '/';
92                 path[strlen(FILEDIR) + 1] = 0;
93                 strcat(path, dir->d_name);
94                 FILE* fileptr = fopen(path, "rb");
95
96                 fseek(fileptr, 0, SEEK_END);
97                 unsigned int size = ftell(fileptr);
98                 fclose(fileptr);
99                 fileptr = fopen(path, "rb");
100
101                 fileWrite(dir->d_name, fileptr, size);
102
103                 fclose(fileptr);
104             }
105             closedir(d);
106         }
107
108         FILE* fileptr = fopen("bin/blob", "wb");

```



```

107         fwrite(pageaddr, BUFSIZE, 1, fileptr);
108         fclose(fileptr);
109     }

```

4.5.4 Executing files

For execution of files, kernel functions were defined as interrupts, such that they could be called by executable files. These interrupts were then included as a part of a standard library inside the crosscompiler folder. The code for one such interrupt pair has been included below.

kernel/libs/interrupts.c

```

1 // readdisksegment(int lba, char sectors) -> void*
2 extern unsigned int _idt80;
3 void idt80() {
4     *(unsigned int*)0x1000 = (unsigned
5         ↪ int)diskReadSector(*(unsigned int*)0x1000,
6         ↪ *(char*)0x1010);
7 }

```

crosscompiler/libs/interrupts.c

```

1 void* diskReadSector(unsigned int lba, unsigned char sectors) {
2     *(unsigned int*)0x1000 = lba;
3     *(unsigned char*)0x1010 = sectors;
4     __asm__ volatile ("int $80");
5     return (void*)(*(unsigned int*)0x1000);
6 }

```

5 Appendix

.1 INT 13₁₆ AH 02₁₆

Register	Value
AL	Number of sectors to read
CH	Cylinder
DH	Head
CL	Sector
DL	Drive
ES:BX	Output offset

.2 Segment Descriptor

Bits	Segment	Value
0-15	(1)	Segment limit
16-39	(1)	Segment base
40-43	(flag)	Type
44	(flag)	S
45-46	(flag)	DPL
47	(flag)	P
48-51	(2)	Segment limit
52	(flag)	A
53	(null)	0
54	(flag)	DB
55	(flag)	G
56-63	(2)	Segment base

.3 INT 15₁₆ EAX E820₁₆

Register	Value
EAX	E820 ₁₆
EBX	0
ECX	24
EDX	534D4150 ₁₆
ES:DI	Output buffer

.4 VGA Options

Value	Type	Resolution	Colourspace
00 ₁₆	Text	40x25	1
01 ₁₆	Text	40x25	16
02 ₁₆	Text	80x25	1
03 ₁₆	Text	80x25	16
04 ₁₆	CGA	320x200	4
05 ₁₆	CGA	320x200	1
06 ₁₆	CGA	640x200	2
07 ₁₆	MDA	80x25	1
0D ₁₆	EGA	320x200	16
0E ₁₆	EGA	640x200	16
0F ₁₆	EGA	640x350	1
10 ₁₆	EGA	640x350	16
11 ₁₆	VGA	640x480	1
12 ₁₆	VGA	640x480	16
13 ₁₆	VGA	320x200	256

.5 VBE1 Functions

Value	Type	Resolution	Colourspace
100 ₁₆	Graphics	640x400	256
101 ₁₆	Graphics	640x480	256
103 ₁₆	Graphics	800x600	256
104 ₁₆	Graphics	1024x768	16
105 ₁₆	Graphics	1024x768	256
106 ₁₆	Graphics	1280x1024	16
107 ₁₆	Graphics	1280x1024	256
10D ₁₆	Graphics	320x200	32K (1:5:5:5)
10E ₁₆	Graphics	320x200	64K (5:6:5)
10F ₁₆	Graphics	320x200	16.8M (8:8:8)
110 ₁₆	Graphics	640x480	32K (1:5:5:5)
111 ₁₆	Graphics	640x480	64K (5:6:5)
112 ₁₆	Graphics	640x480	16.8M (8:8:8)
113 ₁₆	Graphics	800x600	32K (1:5:5:5)
114 ₁₆	Graphics	800x600	64K (5:6:5)
115 ₁₆	Graphics	800x600	16.8M (8:8:8)
116 ₁₆	Graphics	1024x768	32K (1:5:5:5)
117 ₁₆	Graphics	1024x768	64K (5:6:5)
118 ₁₆	Graphics	1024x768	16.8M (8:8:8)
119 ₁₆	Graphics	1280x1024	32K (1:5:5:5)
11A ₁₆	Graphics	1280x1024	64K (5:6:5)
11B ₁₆	Graphics	1280x1024	16.8M (8:8:8)

.6 INT 10₁₆ AX 4F02₁₆

Register	Value
AX	4F02 ₁₆
BX	Mode
BX	4XXX ₁₆ for linear framebuffer

.7 Enabling A20 Line through the Keyboard Controller

Port	Value
64 ₁₆	AD ₁₆
64 ₁₆	D0 ₁₆
60 ₁₆	<i>read</i>
64 ₁₆	D1 ₁₆
60 ₁₆	<i>(read) OR 2</i>
64 ₁₆	AE ₁₆

References

- [1] Compaq Computer Corporation, Phoenix Technologies Ltd, Intel Corporation (1996) *BIOS Boot Specification* pg 29 ch 6.5.1 *Booting from BAIDs*
- [2] Ralf Brown (2000) *Ralf Browns Interrupt List* interrup b int 13
- [3] IDEMA (2013) *The Advent of Advanced Format*
- [4] Intel Corporation (2016) *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 1: Basic Architecture* pg 1.6 ch 1.3.4 *Segmented Addressing*
- [5] Compaq Computer Corporation, Phoenix Technologies Ltd, Intel Corporation (1996) *BIOS Boot Specification* pg 12 ch 3.3 *Devices with PnP Expansion Headers*
- [6] Intel Corporation (2024) *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3A: System Programming Guide, Part 1* pg 2.1 ch 2 *System Architecture Overview*
- [7] Compaq Computer Corporation, Phoenix Technologies Ltd, Intel Corporation (1996) *BIOS Boot Specification* pg 43 ch D.1 *Use DL for Drive Number*
- [8] Intel Corporation (2022) *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2 (2A, 2B, 2C, 2D): Instruction Set Reference, A-Z* pg 4.35 ch 3.3 *Model-Specific Registers*
- [9] Wikipedia (2024) *Global Descriptor Table* ch Description
- [10] Robert Collins (2001) *A20/Reset Anomalies* ch A20/Reset Anomalies
- [11] Wikipedia (2024) *Global Descriptor Table* ch Modern Usage
- [12] OsDev (2024) *Detecting Memory (x86)* ch 2.1
- [13] OsDev (2024) *VGA Fonts* ch 2.4
- [14] Wikipedia (2024) *Mode 13h* ch 2.4
- [15] Video Electronics Standards Association (1998) *VESA BIOS EXTENSION (VBE) Core Functions Standard* pg 17
- [16] Intel Corporation (2016) *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3A: System Programming Guide, Part 1* pg 9.17 ch 9.9.2 *Switching to Protected Mode*
- [17] Intel Corporation (2024) *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2 (2A, 2B, 2C, 2D): Instruction Set Reference, A-Z* pg 3.610 ch 3.3 *Load Global/Interrupt Descriptor Table Register*