

```
<$>: hi
```

```
<$> Hallo Welt!
```



Adrian Stoop

Mein eigenes Operating System für eine x86 CPU

Inhaltsverzeichnis

Projektbeschreibung:	3
Projektplanung	4
Einführung	5
Warum ein Eigenes Operating System (OS) selber programmieren?	6
Was ist ein Betriebssystem?	6
Ablauf wenn man einen PC mit einer x86 CPU startet	6
Die Memory Map	6
Register und Interrupts	7
Der Minikernel	9
Wozu braucht man einen Bootloader?	9
Real Mode:	9
Bootbarer Minikernel	10
Bootloader + nachgeladener Kernel	11
Der C-Kernel	12
16-Bit-Protected Mode	12
32-Bit-Protected Mode	12
Real Mode / Protected Mode Fazit	13
Protected Mode Aktivieren	13
A20-Gate	13
Deskriptortabellen	14
Segmentselektor	15
GDT	15
Sprung zum C-Kernel	17
Farben im Textmodus:	17
Module in C	19
Hilfsmodule	21
Scancodes und ASCII	21
Keyboard Driver	22
Interrupts und Exceptions	23
Interrupt Requests IRQ	24
Exceptions	25

Interrupt Description Table (IDT)	25
Remapping von IRQ.....	27
IRQ Handler	29
System Clock / Programmable Interval Timer (PIT)	29
Keyboardtreiber von Polling auf IRQ1.....	30
Video.c & Keyboard.c wird ausgebaut	30
Systemsfrequenz verändern.....	31
Abschluss Semesterarbeit, begin freizeit Projekt.....	32

Projektbeschreibung:

Projektbeschreibung:

Mein Ziel ist es ein eigenes Operating System für einen x86 CPU Computer.

Editor, Compiler, Linker, x86 Emulator werden übers Internet beschafft.

Editor : Sublime Text 3

Asm >> bin : nasm

Linker/ Compiler : gcc

Emulator : bochs

Ich werde nur Assembler und C benutzen. Am Schluss soll das OS von einer Floppy Disk an einem richtigem PC (x86) bootbar sein.

Folgende Aufgaben soll mein OS können:

- Bootbarer Minikernel
- Bootloader + nachgeladener Kernel
- Auf Instruktionen im RM reagieren (dank BIOS einfache Handhabung)
- A20-Gate und PM aktivieren, GDT/GDTR
- Sprung vom ASM- zum C-Kernel (Lesbarkeit, Module)
- Scansonde und ASCII Tabelle einlesen (Tastatur- Treiber)
- Rudimentäre Textausgabe auf Bildschirm (Video RAM 0xB8000)
- Rudimentäre Texteingabe mit Tastatur ("Polling")
- Auf Interrupt Request (IRQ) soll das OS reagieren. (z.b. System Clock).
- Eine Interrupt Description Table anlegen (IDT)
- IRQ handler um Master PIC und Slave PIC zu setzen und zurücksetzen.
- OS soll eine System Uhr haben die stimmt beim starten(System Clock/ Programmable Interval Timer)
- Rudimentäre Texteingabe mit Tastatur durch interrupts (IRQ)
- Key Queue und Key handlen (Tastenschläge >> Warteschlange(register) bis Programm Tasten braucht)
- Systemfrequenz verändern.
- Einfaches Programm im OS starten.

Projektplanung

Eignes Operating System (WinBex)	11.09.2015	18.09.2015	25.09.2015	02.10.2015	09.10.2015	16.10.2015	23.10.2015	30.10.2015	06.11.2015	13.11.2015	20.11.2015	27.11.2015	04.12.2015	11.12.2015	18.12.2015	25.12.2015	01.01.2016	08.01.2016	15.01.2016
Projektbeschreibung	x																		
Projektplanung	x																		
Doku		x	x	x	x	x	x	x	x	x	x	x	x	x	x	x			x
Presentation (PPP)																	x	x	
Bootbarer Minikernel	x																		
Bootloader + nachgeladener Kernel		x																	
Auf Instruktionen im RM reagieren			x																
A20-Gate und PM aktivieren, GDT/GDTR				x															
Sprung vom ASM- zum C-Kernel					x														
Rudimentäre Textausgabe auf Bildschirm					x														
Rudimentäre Texteingabe mit Tastatur						x													
Auf Interrupt Request (IRQ) reagieren							x												
Interrupt Description Table anlegen								x											
IRQ handler für Master PIC und Slave PIC									x										
System Clock/ Programmable Interval Timer										x									
Rud. Texteingabe mit Tastatur durch interrupts (IRQ)											x								
Key Queue und Key handle												x							
Systemfrequenz verändern													x	x					
Code auskommentieren															x	x	x	x	
Stunden aufwand geschätzt	2	10	15	10	15	10	15	10	10	10	10	10	10	10	10	10	5	5	2
Stunde Aufwand Real	3	15	18	15	20	23	17	20	8	2	14	15	15	12	8	20	15	5	

Einführung

Diese Dokumentation soll als Log-File für dieses Projekt dienen, wie ein kleines Tutorial. Ich habe mir seit einem halben Jahr den Gedanken ein eignes Operating System zu programmieren nicht aus dem Kopf gebracht und möchte zu gern Wissen wie ein Heutiges Betriebssystem aufgebaut ist. Da ich das Risiko meinem eigenen Rechner zu schaden nicht eingehen möchte, benutzte ich einen Emulator.

Meine Informationen habe ich meistens aus dem Internet geholt und zwar von diesen Seiten:

<http://www.lowlevel.eu/wiki/Hauptseite> (Deutsch)

http://wiki.osdev.org/Main_Page (Englisch)

Wichtig: Wer meine Beispiele oder mein OS auf einem Realem PC testen möchte, haftet selber wenn was Kaput geht. Assembler ist eine mächtige Sprache und kann die Hardware nachhaltig beschädigen! Ich empfehle einen x86 Emulator.

Notwendige Tools:

1. Editor für die C-/Asm- Dateien, ich benutze Sublime Text 3.
<http://www.sublimetext.com/3>
2. Programm zum erzeugen von binären Dateien, ich benutze NASM
<http://www.nasm.us/>
3. Emulator um das OS zu testen, ich benutzte Bochs
<http://bochs.sourceforge.net/>
4. Compiler und Linker, ich benutzte DJGPP (nur auf 32-bit Systemen möglich, da man 16-bit Code erzeugen muss und das geht nicht auf 64-bit Rechnern(falscher CPU aufbau))
<http://www.osdever.net/downloads/compilers/DJGPP-Installer-nocpp.exe>
Nach der Installation muss man noch 2 Systemvariable erstellen, damit DJGPP erfolgreich ausgeführt werden kann. Die Zwei Variablen heißen, und haben den wert:

Name: DJGPP

Wert: C:\djgpp\djgpp.env

Name: Path

Wert: C:\DJGPP\bin;

Wenn Sie mit bochs meine Beispiele ausprobieren wollen, müssen Sie in der .bxrc in Zeile 9 den Pfad zur .bin Datei ändern.

```
floppya: 1_44=C:\Ihren Pfad\kernel.bin, status=inserted
```

Warum ein Eigenes Operating System (OS) selber programmieren?

Es gibt gewisse Menschen die möchten gern mehr über den Ablauf eines Rechners erfahren und möchten darum ein eigenes OS erstellen. Nur wenn man diesen Ablauf kennt und auch versteht, also nicht nur lesen sondern auch ausprobieren, weiss man wie ein heutiges OS aufgebaut ist. Es ist eine grosse Herausforderung und am Schluss hat man ein breites Detailwissen über die Funktionen und Hardware seines Rechners. Natürlich kann kein „Hobby OS“ Linux, Windows oder andere OS schlagen, aber das ist ja nicht das Ziel das ich habe.

Was ist ein Betriebssystem?

Was genau ein Betriebssystem ist und was nicht ist eine Schwierige Frage. Ein Betriebssystem kümmert sich im Rahmen der Abstraktion und Schaffung um die Schnittstellen. Es stellt folgende Punkte zur Verfügung:

1. Abstraktion von Geräten und Diensten
2. Application Programming Interface (API)
3. Die Unabhängigkeit von jeweiliger Hardware
4. Verwaltung von Ressourcen wie Zeit, Speicher, Interrupts etc.

Ablauf wenn man einen PC mit einer x86 CPU startet

1. Das Codesegment-Register wird mit dem Wert 0xFFFF geladen. An dieser Adresse startet das BIOS (**B**asic **I**ntput **O**utput **S**ystem) und checkt alle Medien die angeschlossen sind.
2. Das BIOS sucht eine Datei mit der Bootsignatur 0x55AA in den letzten 2 Byte des Sektors. Ein Sektor ist 512 Byte gross. Hat das BIOS dies gefunden ladet es den Code an die Adresse 0x7C00. Dort steht genau dieser Sektor als Speicher zur Verfügung.
3. Im Bootsektor sind Anweisungen den Rest des OS (Kernel) nachzuladen. Der Kernel ist der Kern des Betriebssystems, er kümmert sich um Interrupts, Hardwarezugriffe, Multitasking und die Zusammenarbeit mit den Benutzer-Applikationen.

Die Memory Map

Adressenaufteilung einer x86 CPU

Start Adresse	End Adresse	Grösse	Type	Beschreibung
0x00000000	0x000003FF	1 KB	RAM, besetzt	Interrupt Vektor Table
0x00000400	0x000004FF	256 Byte	RAM, besetzt	BIOS Data Area
0x00000500	0x00007BFF	~30 KB	RAM, verfügbar	Free for use
0x00007C00	0x00007DFF	512 Byte	RAM, verfügbar	Boot-Sector
0x00007E00	0x0007FFFF	480.5 KB	RAM, verfügbar	Free for use
0x00080000	0x0009FBFF	120 KB	RAM, verfügbar	Ffu wenn vorhanden
0x0009FC00	0x0009FFFF	1 KB	RAM, besetzt	Extended BIOS Data Area
0x000A0000	0x000FFFFFFF	384 KB	verschieden	Video Memory

Register und Interrupts

Welche Register gibt es und was sind Interrupts?

Überblick der Register der x86 CPU (16-Bit Version):

	HIGH	LOW
AX	0x00	
BX		
CX		
DX		

Hier sind die sogenannten Mehrzweck-Register. Low und High Register können separat angesprochen werden.

AX	=	AH + AL	Akkumulator
BX	=	BH + BL	Basis register
CX	=	CH + CL	Counter
DX	=	DH + DL	Datenregister für Ein/Aus- gabeoperationen

Weitere 16-Bit Register:

F	<input type="text"/>	<input type="text"/>	IP
SP	<input type="text"/>	<input type="text"/>	CS
BP	<input type="text"/>	<input type="text"/>	DS
SI	<input type="text"/>	<input type="text"/>	SS
DI	<input type="text"/>	<input type="text"/>	ES

BP	Base Pointer	Basisregister
SP	Stack Pointer	Zeiger auf Stack
SI	Source Index	Indexregister für Quell-Operanden
DI	Destination Index	Indexregister für Ziel-Operanden
IP	Instruction Pointer	Steuerregister für Offset-Adresse vom nächsten Befehl
F	Flag Register	Statusregister mit Information (z.B. Carry-bit)
ES	Extra Segment	Arbeitet mit DI
SS	Stack Segment	Arbeitet mit SP
DS	Daten Segment	Segmentregister für weitere Daten
CS	Code Segment	Segmentregister für Instruktionen

Heutige Prozessoren sind immer noch auf dieses Prinzip aufgebaut ausser, dass es echte 32 Bit-Register sind. Man stellt ein E vor den Namen damit man erkennt das es ein 32 Bit Register ist. Es sind Ausserdem noch weitere Register dazu gekommen:

EAX	AX			ESI	SI
	AH	AL			
EBX	BX			EDI	DI
	BH	BL			
ECX	CX			EBP	BP
	CH	CL			
EDX	DX			ESP	SP
	DH	DL			
CS	FS			EFLAGS	FLAGS
DS	GS				
ES	SS			EIP	IP

Es gibt Feste Standard Paarungen der Register:

CS:IP adressiert den Programmspeicher
SS:SP adressiert den Stack (LIFO = LastInFirsOut)

Ein Interrupt unterbricht die CPU, um eine andere Routine auszuführen. Es gibt Zwei Arten von Interrupt und zwar Synchrone und Asynchrone. Synchrone werden nur ausgelöst wenn die CPU gewisse Bedingungen erfüllt und Asynchrone treten nicht bei derselben Stelle auf, sie können nicht vorhersehbar sein.

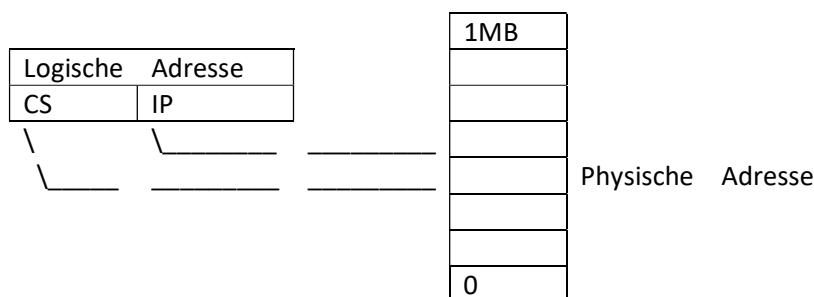
Der Minikernel

Wozu braucht man einen Bootloader?

Da von der Adresse 0x7C00 nur 512 Byte Platz für das OS gegeben ist, sollte man durch den Bootloader den eigentlichen Kernel vom jeweiligen Laufwerk nachladen. Wenn der Kernel nachgeladen ist springt man zu seiner Startadresse. Beim Kernel braucht man keine Boot-Signatur mehr.

Real Mode:

Im Real Mode, eigentlich Real Address Mode, befindet sich die Basisadresse im Segmentregister. Jeder x86 Prozessoren starten nach dem reset im Real Mode. Durch rechnen lässt sich die physische Adresse aus der logischen Adresse Berechnen:



CS = Basisadresse (Segment Selektor)

IP = Offset (Segment Offset)

Physische Adresse = $(16 * \text{Segment Selektor} + \text{Segment Offset})$

Im Real Mode existiert kein Zugriffsschutz. Jedes Programm läuft auf dem Hauptspeicher und hat den ganzen Zugriff auf die Hardware. In diesem Mode haben die Segmentregister nur einen 16 Bit Andersraum und können dadurch nur 1MB Speicher ansprechen. Dies lässt sich berechnen indem man den Inhalt des Basisadressregister (hier CS) mit 16 (0x10 oder einfach 4 stellen nach links shiften) Multipliziert und zur Offsetadresse Addiert. Da die Offset Adresse auch nur 16 Bit beschränkt ist, ergibt sich eine 20 Bit Lineare Adresse. $2^{20} = \sim 1\text{MB}$

Bootbarer Minikernel

Dieser mini Kernel ist so klein das er alleine in den Bootsektor des Speichers passt(unter 512 Byte) und daher keinen Bootloader braucht. Das Ziel dieses Kernel ist auf „hi“ zu reagieren. Bei „hi“ soll der Kernel „Hallo Welt“ auf dem Bildschirm ausgeben. Code:

```
mov ax, 0x07C0
mov ds, ax
mov es, ax

mov si, willkommen_1
call print_string
mov si, willkommen_2
call print_string
```

Zuerst schreiben wir den Wert 0x07C0 ins Allzweckregister (AX) und in die Segmentregister DS und ES.

Danach laden wir die Adresse des willkommen_1 Strings ins Indexregister und rufen die Funktion print_string auf.

```
print_string:
  lodsb

  or al, al
  jz .done

  mov ah, 0x0E
  int 0x10

  jmp print_string

.done:
  ret
```

lodsb (**load string byte**) lädt ein Byte des Strings im SI-Register ins low Register des AKKUs. Wenn der String fertig ist wird nichts in AKKU geladen und somit wird aus dem OR Befehl mit sich selber das Zero-Flag gesetzt. Ist der String noch nicht fertig wird er mit Hilfe des BIOS-Interrupts 0x10 auf dem Bildschirm ausgegeben.

```
strcmp:
.loop:
  mov al, [si]
  mov bl, [di]
  cmp al, bl
  jne .notequal

  cmp al, 0
  je .done

  inc di
  inc si
  jmp .loop

.notequal:
  cld
  ret

.done:
  stc
  ret
```

strcmp (**String compare**) vergleicht den eingegeben Befehl mit den vorhandenen Befehlen des Kernels, sind sie gleich so wird das Carry-Bit gesetzt (stc) und zurückgesprungen. Durch inc kann Byte für Byte verglichen werden. Ist der Befehl nicht gleich, so wird das Carry-Bit nicht gesetzt (clc) und zurückgesprungen.

```
times 510-($-$$) db 0
db 0x55
db 0xAA
```

\$ und \$\$ sind spezielle NASM Befehlssätze. \$ ersetzt die aktuelle Programm Zeile und \$\$ die Startadresse (hier 0x7C00). Da die min. Sektorgröße von 512 Byte erreicht werden muss und in den letzten 2 Byte's noch die Bootsignatur gesetzt werden muss, kann man mit dieser Zeile an die letzten beiden Stellen des Sektor Springen. Mit db (define Byte) kann man die Signatur dann setzen.

Um das Ganze zu testen muss man mit NASM die binäre Datei erstellen. NASM.exe sollte in dem Ordner sein wo auch die asm Datei ist. Mit shift und rechtecklick im Ordner kann man ganz einfach die Eingabeaufforderung hier öffnen. Folgender Befehl erstellt die binäre Datei:

```
nasm kernel.asm -f bin -o kernel.bin
```

In jedem meiner Ordner ist eine make.cmd Datei die diese Aufgabe übernimmt:

```
@echo off
del kernel.bin
nasm kernel.asm -f bin -o kernel.bin
pause
exit
```

in der bochs config datei, die sich auch in diesem Ordner befindet sollte man den Pfad zur kernel.bin Datei anpassen, sonst kommt ein Error:

```
floppya: 1_44=C:\>>der pfad zu dieser datei<<\kernel.bin, status=inserted
```

Bootloader + nachgeladener Kernel

Durch die Bootsignatur wird der Bootloader automatisch eingelesen. Doch wie kann man den Kernel nachladen? Ganz einfach mit dem BIOS-interrupt 0x13, der Read Funktion.

Parameter:

AH	0x02
AL	Wie viele Sektoren einlesen?
CX	Cylinder + Sektor
DH	Head
DL	Driver, Laufwerk
ES:BX	Ziel Adresse, wo hin laden...

Im Code sieht das so aus:

```
;-----;
; Lese Kernel vom Floppy ein:
;-----;

load_kernel:
    xor ax, ax                ; "reset" vom AKKU (XOR)
    int 0x13                 ; Laufwerk zurücksetzen
    jc load_kernel           ; fehler? noch mal (zurückspringen)

    mov bx, 0x8000           ; Setzte startadresse vom Kernel

;-----;
; Parameters für lese Funktion (inrupt 0x13)
;-----;
    mov dl,[bootdrive]       ; wähle Laufwerk aus (boot und kernel sind auf dem gleichem)
    mov al,10                 ; lese 10 Sektoren
    mov ch, 0                 ; cylinder = 0
    mov cl, 2                 ; sector = 2
    mov dh, 0                 ; head = 0
    mov ah, 2                 ; funktion "lese"
    int 0x13                 ; Bios funktion
    jc load_kernel           ; fehler? noch mal (zurückspringen)
```

Um das Ganze zu testen muss man jetzt 2 binäre Dateien erstellen und die dann zusammenkopieren:

```
nasm boot.asm -f bin -o boot.bin
nasm kernel.asm -f bin -o kernel.bin
copy /b boot.bin + kernel.bin WinBex.bin
```

Der C-Kernel

16-Bit-Protected Mode

Bei diesem Mode wird für die Basis und die Segmentauswahl eine Deskriptortabelle zwischen die logische und physische Adresse eingeschoben. Diese Tabelle funktioniert wie ein Zeiger. Aus der Logischen Adresse wird mit Hilfe dieser Tabelle eine Physische Adresse angesprochen. Diese Adresse kann nicht berechnet werden ausser man kennt den Inhalt der Tabelle.

32-Bit-Protected Mode

Ab den 386er wurde der Protected Mode auf 32 Bit erweitert. Es gibt auch einen neuen „Paging“ Mechanismus. Der physische Speicher teilt sich in „Seiten“ (Pages) zu jeweils $2^{12} = 4\text{KB}$. Die Berechnung einer physischen Adresse erfolgt nun aus der linearen Adresse noch über diesen Paging-Schritt. Die 32 Bit breite lineare Adresse wird dabei zweigeteilt. Die höherwertigen 20 Bit verweisen auf eine Speicherseite. Die niederwertigen 12 Bit sind der Seiten-Offset der selektierten Seite. Bit 31 ...12 werden also als Index in eine sogenannte Pagetable verwendet. Diese Pagetable enthält die physische Basisadresse der jeweiligen Seite.

Real Mode / Protected Mode Fazit

- Real Mode = Zeiger auf physische Adresse im Speicher (z.b. CS:IP)
- 16 bit Pro. Mode = Zeiger auf Zeiger (Deskriptor -> Phys. Basisadresse)
- 32 bit Pro. Mode = Zeiger auf Zeiger auf Zeiger (Selektor -> Deskriptor -> Page -> Phys. Basis)

im Protected Mode spricht man eine virtuelle Adresse an.

Man hat also keine aus den logischen Adresswerten berechenbare reale Adresse wie im Real Mode.

Dies erfolgt durch Entkoppelung mittels Indizes (Zeiger, Pointer) der Descriptor Table und Page Table.

Protected Mode Aktivieren

Um in den Protected Mode zu schalten muss man mindestens folgende Dinge haben:

1. Global descriptor table (GDT) anlegen
2. Laden des GDTR durch die GDT Adresse
3. PM (Protected Mode) Aktivieren durch setzen von Bit 0 des Registers CR0 (PE-Bit)
4. "FAR_JUMP" durchführen zum Lehren der Warteschlange
5. A20-Leitung (A20-Gate) Aktivieren, damit kein 8088-kompatibler "wrap" der Speicher-Adressierung erfolgt.

A20-Gate

Das A20-Gate steuert die 21. Adressleitung der CPU.

Heutige CPU's besitzen 32 Adressleitungen, da man aber im Real Mode nur ~1MB ansprechen kann wurde diese Leitung bei starten des Rechners deaktiviert. Wozu?

Durch die Physische Adressberechnung kommt man bei FFFF:FFFF auf 0x10FFEF = 1114095 Byte.

Da aber 1 MB nur 1048576 Byte gross ist kann man Adressen ansprechen die es gar nicht gibt. Anstatt einen Fehler auszugeben wird die CPU einfach wieder bei 0 beginnen, also ist die Adresse von FFFF:FFFF nicht 0x10FFEF sondern 0x0FFEF. Dies nennt man Wrap-Around Effekt.

Zu Zeiten von DOS hat man diesen Wrap-Around ganz gezielt für gewisse Programme genutzt.

Was würde nun aber passieren, wenn wir dieses DOS-Programm, das ja damit rechnet, dass es oberhalb von 1 MB einen Wrap-Around gibt, in einem 386er laufen lassen, der ja bekanntlich nicht mehr nur 20, sondern 32 Bit für den Adressbus hat? In diesem Fall würde ja ein Speicherbereich oberhalb von 1 MB existieren und das DOS-Programm würde womöglich einen Fehler machen, der im ersten Augenblick wohl gar nicht auffallen würde, aber am Ende vielleicht fatal wäre.

Und um der Sache Abhilfe zu schaffen, haben die IBM-Entwickler eine Möglichkeit geschaffen, die 21te Adressleitung abschalten zu können. Somit würde es bei 1 MB wieder einen Wrap-Around geben und alle Programme wären zufrieden. Und da beim Booten des PCs der Prozessor von Anfang an eh im Real-Mode arbeitet, ist auch die 21te Adressleitung (A20) von Anfang an abgeschaltet.

Also muss man Software technisch diese Adressleitung öffnen bevor man in den PM wechselt.

Es gibt mehrere Wege wie man diese Leitung öffnet:

1. Keyboard Controller (in meinem Bootloader vorhanden ab 0.00.03)
2. Modernes BIOS
3. System Control an Port A (in meinem Bootloader vorhanden ab 0.00.03)

Deskriptortabellen

Es gibt 3 Arten von Deskriptortabellen:

Global descriptor table (GDT)	=	für alle Task identisch
Local descriptor table (LDT)	=	jeder Task kann eine eigene LDT haben
Interrupt descriptor table (IDT)	=	wird für Interrupt Service Routinen verwendet

Diese Tabellen liegen im Hauptspeicher und werden vom Betriebssystem verwaltet. Im speziellen Registern werden die Basisadressen gespeichert:

GDTR	=	speichert die physische Basisadresse der GDT (24 Bit)
IDTR	=	speichert die physische Basisadresse der IDT (24 Bit)
LDTR	=	ist ein 16-Bit Segmentselektor für das Segment, in dem die LDT liegt.

Jeder dieser Deskriptortabellen enthält maximal 8192 Einträge, sogenannte Segmentdeskriptoren. Jeder Segmentdeskriptor beschreibt ein Speichersegment und enthält dafür folgende Felder:

1. Die Segmentlänge (max. 64KB)
2. Die Startadresse (auf 1 Byte genau)
3. Segmenttyp (S=0 Systemsegment, S=1 Benutzersegment)
4. Descriptor Privilege lvl (0 = höchste Rechte, 3 niedrigste)
5. Present (ist P-Bit auf 1, ist Segment im Hauptspeicher angelegt)

* Der erste Eintrag (mit Index 0) der GDT ist als Nullselektor reserviert und darf auf keinen Speicherbereich verweisen.

Segmentelektor

Im Real Mode enthalten die Segmentregister direkt die Startadresse des Speichersegments (genauer: die obersten 16 Bit der 20-Bit-Real-Mode-Speicheradresse). Im Protected Mode dagegen enthalten die Segmentregister einen Verweis auf eine der beiden Deskriptortabellen (GDT oder LDT), in denen die Eigenschaften der Speichersegmente festgehalten sind.

GDT

Zuerst mal den Aufbau der Deskriptoren:

	High Byte	Low Byte
Word 3	Basisadresse (Bit 24-31)	Flags
Word 2	Zugriffsrecht	Segm. Länge (Bit 16-19)
Word 1	Basisadresse (Bit 0-15)	
Word 0	Segmentlänge (Bit 0-15)	

Ein Deskriptor umfasst 8 Byte. Nachfolgend findet man diesen Typ als CODE-Deskriptor (CODE_Desc) und DATA-Deskriptor (DATA_Desc). Das erste Word (2 Byte) umfasst die Segmentlänge, dann kommt das zweite Word mit der Basisadresse. Das dritte Byte der Basisadresse findet sich im Anschluss als fünftes Byte des Deskriptors.

Die Zugriffsrechte werden mit 8 Bits angegeben:

Bit	Bedeutung	Bei 0	Bei 1
7	Present Bit (P)	Descriptor ist undefiniert	Descriptor besitzt einen Basiswert und ein Limit
6	DPL* (High)	-	-
5	DPL* (Low)	-	-
4	Segment Bit (S)	System Descriptor	Code, Data oder Stack Descriptor
3	Type	-	-
2	Type	-	-
1	Type	-	-
0	Accessed Bit (A)	Segment hat keinen Zugriff	Segment hat Zugriff

*DPL = Deskriptor Privileg Level (0 = 00b, 1 = 01b, 2 = 10b, 3 = 11b)

Die Drei Type Bits können 8 Möglichkeiten besitzen:

000	Data read only
001	Data read/write
010	Stack read only
011	Stack read/write
100	Code execute only
101	Code execute /read
110	Code execute only, conforming
111	Code execute /read, conforming

Die Beschreibung der GDT erfolgt im **GDTR (Global Descriptor Table Register)**. Dieses besteht aus sechs Byte (48 Bit). Die ersten beiden niederwertigen Byte beschreiben die Länge der GDT. Die Zahl der Deskriptoren ergibt sich hieraus durch $\text{Anzahl} = \text{Länge GDT} / 8$. In unserem konkreten Fall ist dies $24/8 = 3$. Anschließend folgen vier Byte (32-Bit-Adresse), die die Basisadresse der GDT enthalten, in unserem Fall beginnt die Tabelle beim Label `NULL_Desc`.

```
gdtr:
Limit  dw 24      ; length of GDT
Base   dd NULL_Desc ; base of GDT ( linear address: RM Offset + Seg<4 )
```

Der erste Deskriptor muss ein **Null-Descriptor** sein:

```
NULL_Desc:
dd 0
dd 0
```

Im **CODE-Descriptor** sieht's schon spannender aus.

Die Segmentlänge beträgt `0xFFFF`.

Die Segmentbasis ist `0x0`

Die Zugriffsrechte dekodieren sich wie folgt:

Deskriptor besitzt eine gültige Basis und ein gültiges Limit (Bit 7)

DPL: Privileg 0 (Bit 6, Bit 5)

S: Code, Data, Stack (Bit 4)

Typ: 101 "Code execute/read" (Bit 3, Bit 2, Bit 1)

A: Segment not accessed (Bit 0)

```
CODE_Desc:
dw 0xFFFF ; segment length bits 0-15 ("limit")
dw 0       ; segment base byte 0,1
db 0       ; segment base byte 2
db 10011010b ; access rights
db 00001111b ; bit 7-4: 4 flag bits granularity, default
```

Im **DATA-Descriptor** ist alles analog aufgebaut. Der Unterschied besteht bei den Zugriffsrechten.

Sprung zum C-Kernel

Zuerst setzt man den Sprungpunkt vom Real Mode auf Global, damit man von überall hin springen kann. Natürlich darf die Deklaration [extern _main] nicht fehlen. Mit dieser Zeile kann man dann "Rauspringen" zur Hauptfunktion Main, die dann in C geschrieben ist. Nun muss man das A20-Gate öffnen mit dem PE-Bit im CR0 Register.

Warum wird dieses "Underscore" für "_main" verwendet? In C deklariert man es lediglich als "main". Der Grund ist, dass der Compiler gcc verwendet den Underscore vor allen Funktions- und Variablennamen. Daher muss man einen Underscore hinzufügen, wenn man eine Funktion im C-Code seitens Assemblercode anspricht.

Nach anlegen einer GDT und einem Linkskript für den Compiler sollte der Kernel dann in die Main Funktion des C Files springen.

Ich habe eine Externe GDT genommen, damit ich hier nicht zu viel Zeit verliere und mich versichern kann, dass bei dieser GDT kein Fehler vorhanden ist.

Linkskript:

```
OUTPUT_FORMAT("binary")
ENTRY(RealMode)
SECTIONS
{
    .text 0x8000 : { *(.text) }
    .data : { *(.data) }
    .bss : { *(.bss) }
}
```

Farben im Textmodus:

Im C Kernel hat man aber nicht alle Funktionen vorhanden wie man es kennt. Printf gibt es hier nicht ausser man schreibt die Funktion selber, dass ich auch gemacht habe. Damit diese Funktionen gehen muss man die Hardware in den Funktionen Ansprechen. Wenn man einen Text ausgeben möchte wäre das von der Adresse 0xB8000 bis 0xBC000. Da ich noch im Textmodus bin, muss ich keine eigene Buchstaben "Zeichnen". Der Bildschirm in diesem Mode unterstützt nur Textmodus und kann keine einzelne Pixel ansteuern, nur Zeichen von der Tastatur. Man hat 25 Zeilen an 80 Spalten platz(2000 Zeichen). Der Text an sich kann mit Farbe ausgegeben werden. Ab der Adresse 0xB8000 stehen immer 2 Byte für ein Buchstabe, 1 Byte für Welchen es ist und 1 Byte für die Farbe.

Wie man die Farbe bestimmen kann sieht man in dieser Legende:

Hintergrundfarbe(1) Vordergrundfarbe(2) Blinken/Intensität(3) Intensitätsbit(4)

3	1	1	1	4	2	2	2
---	---	---	---	---	---	---	---

FarbenCode:

0	0	0	Schwarz
0	0	1	Blau
0	1	0	Grün
0	1	1	Türkis
1	0	0	Rot
1	0	1	Magenta
1	1	0	Braun
1	1	1	Hellgrau

Ist das Intensitätsbit oder Blinken/Intensität auf 1 sieht die Tabelle so aus:

0	0	0	Dunkelgrau
0	0	1	Hellblau
0	1	0	Hellgrün
0	1	1	Helltürkis
1	0	0	Hellrot
1	0	1	Hellmagenta
1	1	0	Gelb
1	1	1	Weiss

Um den Bildschirm zu löschen müsste die Funktion etwa so aussehen:

```
void k_clear_screen()
{
    char* vidmem = (char*) 0xb8000;
    unsigned int i=0;
    while(i<(80*2*25))
    {
        vidmem[i] = ' '; //leerschlag
        ++i;
        vidmem[i] = 0x07; //weiss auf schwarz
        ++i;
    };
};
```

Folgende Funktionen hat mein C Kernel bis jetzt:

- K_clear_screen: löscht Bildschirm
- K_printf(nachricht, Linie): Schreibt Nachricht auf Bildschirm, beginnt neue Linie mit \n. Teilt Nachricht wenn Ende der Zeile erreicht ist auf eine Neue Zeile.
- Outportb(port,wert): Sendet einen Wert an die 65536 möglichen Ports.
- Update_cursor(Zeile, Spalte): Setzt „blinkenden“ Cursor an Position Zeile/Spalte.

Da ich jetzt C benutze muss ich nun auch den gcc Compiler einbinden, dass Make file sieht nun so aus:

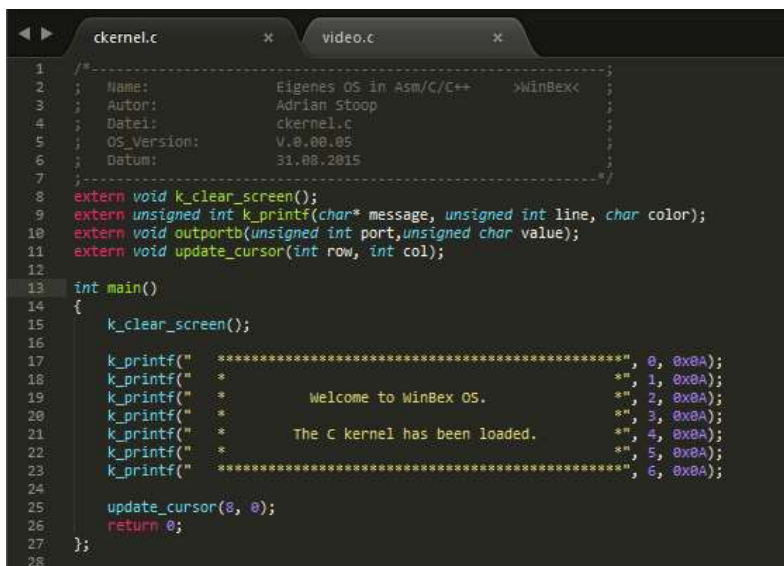
```
nasm -O32 -f bin boot.asm -o boot.bin
nasm -O32 -f aout kernel.asm -o kernel.o
gcc -c ckernel.c -o ckernel.o
ld -T kernel.ld kernel.o ckernel.o
rename a.out ckernel.bin

copy /b boot.bin + ckernel.bin WinBex.bin
```

Module in C

Es gibt keinen Sinn alle Funktionen in ein einziges c File zu packen. Man teilt in C daher das Programm in selbständig übersetzbare Programmeinheiten auf. Diese Aufteilung hat typischerweise logische zusammenhänge der einzelnen Funktionen, die auch durch andere Programmteile genutzt werden können. Diese Programmeinheiten vom Typ xyz.c nennt man Module. Ein Modul kann nicht nur Funktionen, sondern auch andere Datentypen "exportieren". Damit ein Modul Ressourcen aus anderen Modulen "importieren" kann verwendet man das Schlüsselwort extern. Man deklariert also die nicht im Modul befindlichen Funktionen mit dem vorgestellten Schlüsselwort extern.

Nun werde ich das ckernel.c File in mehrere kleinere Files unterteilen. Alles was mit dem Video RAM zu tun hat, kommt ins video.c File. Alle Funktionen müssen aber dann im ckernel.c mit einem extern(bekannt machen) deklarieren.



```
1  /*-----*/
2  ; Name:      Eigenes OS in Asm/C/C++  >WinBex<
3  ; Autor:    Adrian Stoop
4  ; Datei:    ckernel.c
5  ; OS_Version: V.0.00.05
6  ; Datum:    31.08.2015
7  ;-----*/
8  extern void k_clear_screen();
9  extern unsigned int k_printf(char* message, unsigned int line, char color);
10 extern void outputb(unsigned int port,unsigned char value);
11 extern void update_cursor(int row, int col);
12
13 int main()
14 {
15     k_clear_screen();
16
17     k_printf(" *****", 0, 0x0A);
18     k_printf(" *", 1, 0x0A);
19     k_printf(" *      Welcome to WinBex OS.      ", 2, 0x0A);
20     k_printf(" *", 3, 0x0A);
21     k_printf(" *      The C kernel has been loaded.    ", 4, 0x0A);
22     k_printf(" *", 5, 0x0A);
23     k_printf(" *****", 6, 0x0A);
24
25     update_cursor(0, 0);
26     return 0;
27 };
```

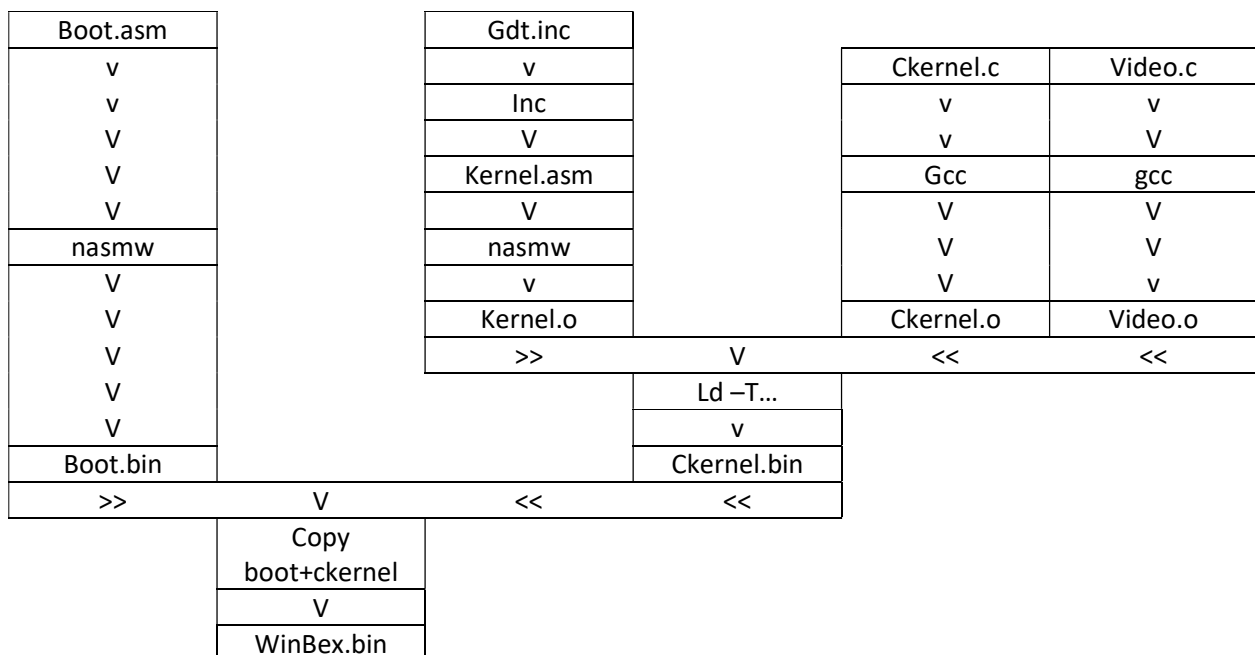
Da jetzt ein neues File dazugekommen ist, muss das make File auch geändert werden:

```
@echo off
del WinBex.bin

nasm -O32 -f bin boot.asm -o boot.bin
nasm -O32 -f aout kernel.asm -o kernel.o
gcc -c ckernel.c -o ckernel.o
gcc -c video.c -o video.o
ld -T kernel.ld kernel.o ckernel.o video.o -o ckernel.bin --verbose
copy /b boot.bin + ckernel.bin WinBex.bin
pause
del video.o
del kernel.o
del ckernel.o
del boot.bin
del ckernel.bin
pause
exit
```

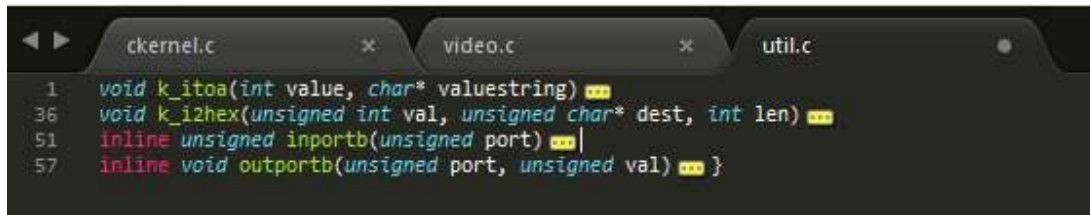
Mein OS besteht aus ganz vielen kleinen Einheiten und muss mittels Tools assembliert, kompiliert, gelinkt und transferiert werden.

Hier eine kleine Grafik, damit der Zusammenhang klar ist:



Hilfsmodule

Wenn ich Zahlen als Text auf dem Bildschirm ausgeben möchte, müssen diese in sogenannte Stringformate umgewandelt werden. Diese Funktionen der Routinen benötige ich immer wieder. Daher erstelle ich die util.c für Hilfsroutinen.



```
1 void k_itoa(int value, char* valstring) {}
36 void k_i2hex(unsigned int val, unsigned char* dest, int len) {}
51 inline unsigned inportb(unsigned port) {}
57 inline void outportb(unsigned port, unsigned val) {}
```

Beide Funktionen müssen auch wieder mit einem extern im kernel.c deklariert werden.

k_itoa = wandelt integer Text in ein Dezimalformat

k_i2hex = wandelt integer Text in ein hexadezimalformat

inportb = ruft einen Wert(Byte) von einem Port ab.

Outportb = gibt einen Wert(Byte) an den Port.

Scancodes und ASCII

Eine PC Tastatur ist eigentlich ein eigenständiger „Computer“, d.h. ein Microcontroller überwacht (scant) ständig die Bestätigung der Tasten. Auch wenn der PC beschäftigt ist, gehen keine Anschläge verloren. Die Programmlogik der Tastatur kümmert sich durch geeignete Scanzzeiten (im 10 Millisekunden-Bereich) um das Unterdrücken der Tastenprellung (keybounce). Zusätzlich muss auch das andauernde Drücken einer Taste verarbeitet werden.

Wird eine Taste gedrückt, so wird der entsprechende Scancode gesendet. Die Scancodes helfen, die physischen Tasten von den länderspezifischen Belegungen zu koppeln. Erst im Keyboard Treiber wird dem Scancode eine konkrete Bedeutung gemäß dem ASCII-Code mittels Keymaps zugeordnet. Durch die Shift-Taste entsteht eine Doppelbelegung vieler Tasten (Groß-/Kleinbuchstaben, Zahlen, Satz- und Sonderzeichen), weshalb man entsprechende Keymaps für die Shift- und die Non-Shift-Tastaturbelegung anbieten muss. Die Tastatur erzeugt übrigens zwei Scancodes pro Tastenanschlag, nämlich einen "down code" beim Drücken und einen "up code" beim Loslassen der Taste.

Heute hat man sich geeinigt, dass alle gängigen OS das Scanset 2 mit eingeschalteter Umwandlung einsetzen. Der Mikrocontroller in der Tastatur überträgt den Scancode mit dem Scanset 2 zum Keyboard Controller auf dem Mainboard. Dieser setzt diesen Code um auf das Scancode Set 1.

Kommt ein Scancode zum PC, so empfängt der Keyboard Controller den Scan Code (Set 2), wandelt diesen um (Set 1), stellt diesen Code am I/O Port 0x60 zur Verfügung und sendet einen Interrupt an die CPU. Das Resultat wandert in den Tastaturpuffer, der normalerweise 16 Plätze hat, **wobei ein Überlauf möglich ist**. Nun ist im Tastaturpuffer nur die Tastennummer abgespeichert, welche gedrückt worden ist. Es ist aber nicht ein Zeichen aus der ASCII-Tabelle! Die Verknüpfung findet erst in der Software des PC-Betriebssystems mittels Keymaps statt.

ASCII(American Standard Code for Information Interchange) stellt einen 7-Bit-Code (dezimal: 0 ...127) für Zeichen dar, der 1968 durch ANSI standardisiert wurde. Nachfolgend eine kompakte tabellarische Darstellung mit Hexzahlen:

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HAT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	EXC	FS	GS	RS	US
2	SP	!	"	#	\$	%	&	'	()	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{		}	~	DEL

Keyboard Driver

Um jetzt etwas im OS einzugeben, muss noch ein Keyboard-„Treiber“ und die Entsprechende Befehle im OS programmiert werden.

Um was einzulesen benötigen wird den Wert von Port 0x60:

```
unsigned int FetchScancode()
{
    return( inportb(0x60));
}
```

Der Scancode wird zuerst überprüft ob die Shift-Taste gedrückt ist. Ob die Taste gedrückt ist oder war sagt uns das 7 Bit. Für den Zustand der Shift-Taste verwenden wir eine Variable, die den aktuellen Zustand der Shift-Taste vor dem von der Funktion zurück gelieferten Scancode der gedrückten Taste speichert. Der Zustand der Shift-Taste wird in der Variable ShiftKeyDown und der Scancode in der Variable scancode festgehalten.

```

unsigned int FetchAndAnalyzeScancode()
{
    unsigned int scancode; // variable für die "roh" taste
    while(1) // endlos, solange keine taste gedrückt wurde
    {
        // warte bis taste gedrückt ist
        while ( !(inportb(0x64)&1) ); // 0x64: lese das statusregister
        scancode = FetchScancode();

        if ( scancode & 0x80 ) //Taste nicht mehr gedrückt? CheckE bit 7 (10000000b = 0x80)
        {
            scancode &= 0x7F; // taste wurde losgelassen, vergleiche nur die 7 tiefsten bits
            if ( scancode == KRLEFT_SHIFT || scancode == KRRIGHT_SHIFT ) // Taste wurde losgelassen
            {
                ShiftKeyDown = 0; // ja, non shift
                continue; // Loop
            }

            // Taste wurde gedrückt. ist shift-Taste auch gedrückt?
            if ( scancode == KRLEFT_SHIFT || scancode == KRRIGHT_SHIFT )
            {
                ShiftKeyDown = 1; // ja, benutzte asclishoft
                continue; // Loop, damit kein scancode mit shift nachgesendet wird.
            }
            return scancode;
        }
    }
}

```

Nun benötige ich eine Keymap(Array), um ASCII zu erzeugen, diese ist in der Datei keyboard.h zu finden.

KRLEFT_SHIFT und KRRIGHT-SHIFT sind auch dort zu finden.

Interrupts und Exceptions

Die Ausgabe auf dem Bildschirm ist in der Regie der CPU, aber bei der Tastatureingabe musste sich die CPU zum Affen machen. Nur durch ständiges Nachschauen im "Briefkasten", also im Tastaturpuffer, erhielt die CPU ihre Informationen. Dieses "Polling" (engl. abfragen) ist aber völlig ineffizient, so dass hier die "Klingel" erfunden werden musste.

Nun wird im Protected Mode ein neues Kapitel aufgeschlagen. Die CPU lässt sich von ihrem Umfeld "unterbrechen" um "ungefragt" Informationen entgegen zu nehmen. Dies nennt man "Interrupts"(klingel). Diese Unterbrechungen waren nur im Real Mode direkt möglich, aber da ich im PM bin, muss ich diese Technik zuerst wieder aufbauen(programmieren).

Interrupt Requests IRQ

Zuerst möchte ich die Meldungen der Zentraleinheit umgebenen Hardware erfassen und darauf reagieren. Die klassische "Systemuhr" (system clock) erzeugt standardmäßig Ticks im Abstand von ca. 18,222 Millisekunden und liefert diese Information als **IRQ0**, das heisst Interrupt Request Nr. 0. Die Tastatur meldet sich bei leerem Tastaturpuffer und Bereitstellung eines neuen Zeichens mit **IRQ1**.

IRQ0	System Clock (Tickt alle 18,222ms)
IRQ1	Tastatur
IRQ2	Programmierbarer Interrupt-Controller
IRQ3	Serielle Schnittstelle COM2
IRQ4	Serielle Schnittstelle COM1
IRQ5	Frei (oft Soundkarte)
IRQ6	Diskettenlaufwerk
IRQ7	Parallele Schnittstelle LPT1
IRQ8	Echtzeitsystemuhr
IRQ9	Frei
IRQ10	Frei
IRQ11	Frei
IRQ12	PS/2-Mouse
IRQ13	Mathematischer Coprozessor
IRQ14	Primärer IDE-Kanal
IRQ15	Sekundärer IDE-Kanal

Man kann Interrupts "remappen", das bedeutet, dass man eine IRQ-Nummer auf eine andere IRQ-Nummer umlenkt.

Üblich ist, dass man im Protected Mode den Bereich von IRQ0 bis IRQ15 dem Bereich von IRQ32 bis IRQ47 zuordnet.

Diese Vorgehensweise werde ich später übernehmen.

Exceptions

Neben den Hardware-Interrupts gibt es sogenannte Exceptions (bedingt durch Programmfehler), also „Ausnahmen“, die durch Fehler verursacht werden. Folgende Fehlermeldungen werde ich verwenden (aus dem Lowlevel wiki):

```
unsigned char* exception_messages[] =
{
    "Division By Zero",      "Debug",                "Non Maskable Interrupt",  "Breakpoint",
    "Into Detected Overflow", "Out of Bounds",        "Invalid Opcode",          "No Coprocessor",
    "Double Fault",         "Coprocessor Segment Overrun", "Bad TSS",                 "Segment Not Present",
    "Stack Fault",          "General Protection Fault",   "Page Fault",              "Unknown Interrupt",
    "Coprocessor Fault",    "Alignment Check",          "Machine Check",           "Reserved",
    "Reserved",             "Reserved",                 "Reserved",                "Reserved",
    "Reserved",             "Reserved",                 "Reserved",                "Reserved",
    "Reserved",             "Reserved",                 "Reserved",                "Reserved"
};
```

Wenn wir in einem Programm ausversehen durch 0 Dividieren (was in der Mathematik verboten ist) soll eine Entsprechende Exception Nr. 0 bei der CPU anfallen. Dann wird das OS „Division By Zero“ ausgegeben und ich muss das dann geeignet handeln.

Interrupt Description Table (IDT)

Genau wie bei der GDT arbeitet man auch bei der Suche nach einem Handler für einen IRQ mit einer Zeigertabelle auf entsprechende Speicherbereiche. Dies nennt sich Interrupt Descriptor Table (**IDT**). Ein sogenanntes Interrupt Descriptor Table Register (**IDTR**) verweist auf die Basis und das Limit (Größe) dieser Tabelle. Wie ist die IDT nun aufgebaut? Wir benötigen dort 256 Einträge für die maximal möglichen IRQ Nummern. Bei einem IRQ ohne Eintrag in der IDT **stürzt die CPU ab**.

Ein Eintrag im IDT sieht so aus:

	High Byte	Low Byte
Word 3	Basisadresse (bit 16-31)	
Word 2	Flags	Immer 0x00
Word 1	Selektor	
Word 0	Basisadresse (bit 0-15)	

Bei den Flags ist die Aufteilung analog zum GDT, allerdings ist das Type Flag 4 Bit breit.

Bit	Bedeutung	0	1
7	P (Present Bit)	Descriptor ist unbestimmt	Descriptor enthält ein gültige basis und limit
6	DPL (High)	Sihe unten	Siehe unten
5	DPL (Low)	Sihe unten	Sihe unten
4	S (Segment Bit)	System Descriptor	Code, Data oder Stack Descriptor
3	Type	Sihe unten	Sihe unten
2	Type	Sihe unten	Sihe unten
1	Type	Sihe unten	Sihe unten
0	Type	Sihe unten	Sihe unten

DPL (Descriptor Privilege Level) setzt das Privileg-Level:

Level 0	00b
Level 1	01b
Level 2	10b
Level 3	11b

Type (in Flags)[b]	bedeutung
0000	-
0001	80286-TSS
0010	Local Descriptor Table (LDT)
0011	Aktives 80286-TSS
0100	80286 Call Gate
0101	Task Gate
0110	80286 Interrupt Gate
0111	80286 Trap Gate
1000	Reserviert
1001	80386-TSS
1010	Reserviert
1011	Aktives 80386-TSS
1100	80386 Call Gate
1101	Reserviert
1110	80386 Interrupt Gate
1111	80386 Trap Gate

Wie man aus der Tabelle herauslesen kann ist der Interrupt Gate Type, den ich brauche für die IDT nur ein Sonderfall von 16 möglichen Gate-Typen. Durch die DPL beschränkt sich der Zugriff auf den Deskriptor. Darum nennt man es auch Gate, man braucht min. die gleiche Privilegstufe wie das Gate als Aufruf durch ein Programm oder Task. Bei geringerer Privilegstufe, reagiert der Prozessor schon bei dem Versuch, das "höherrangige" Gate zu benutzen, mit einer Exception.

Das Present Flag besitzt die gleiche Funktion wie in der GDT. Ist es gesetzt, so befindet sich das beschriebene Segment Aktuell im Speicher.

Mit der folgenden Funktion in C kann man durch asm die entsprechende IDT register füllen in der CPU:

```
static void idt_load()
{
    asm volatile("lidt %0" : "=m" (idt_register));
}
```

Beim Initialisieren der IDT werden 256 Einträge vorgenommen und zunächst komplett alle Werte auf 0 gesetzt.

Um einen Eintrag in die IDT einzufügen, benötigt man folgende Funktion und Parametern:

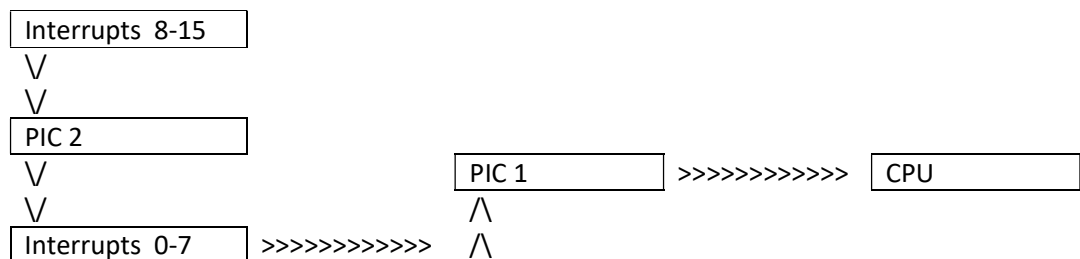
```
void idt_set_gate(unsigned char num, unsigned long base, unsigned short sel, unsigned char flags)
{
    idt[num].base_lo = (base & 0xFFFF);
    idt[num].base_hi = (base >> 16) & 0xFFFF;
    idt[num].sel = sel;
    idt[num].always0 = 0;
    idt[num].flags = flags;
}
```

Ich erstelle ein Array um 16 eigene IRQ-Handler zuordnen zu können. Sozusagen als Behandlungsroutinen im Falle des Auftretens eines IRQ.

```
void* irq_routines[16] =
{
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
};
//Implementiert eine benutzerdefinierte IRQ -Handler für das angegebene IRQ
void irq_install_handler(int irq, void (*handler)(struct regs* r)) {irq_routines[irq] = handler;}

//löscht den benutzerdefinierte IRQ -Handler
void irq_uninstall_handler(int irq) {irq_routines[irq] = 0;}
```

Remapping von IRQ



Am Anfang gab es nur einen PIC mit acht Eingängen. Später wurde ein zweiter PIC hinzugefügt, der mit Eingang 2 des ersten PIC fest verbunden wurde. Heute wird dies nicht mehr so aufgebaut, aber die Methodik ist aus Kompatibilitätsgründen die gleiche geblieben.

Diese PIC haben nur jeweils einen eigenen Port für Daten und Befehle:

		Command	Data
PIC 1	Master	0x20	0x21
PIC 2	Slave	0xA0	0xA1

Im Protected Mode stört die Überlappung der IRQ 0-7 mit möglichen Software-Interrupts. Daher wird alles um 32 Einträge weiter geschoben. Mit den Command und Data Angaben oben, kann man nun eine Funktion schreiben die den Data-Offset des Master PIC auf 0x20(32) und des Slave PIC auf 0x28(40)setzt:

```
void irq_remap(void)
{
    // sage PICs mit 0x11 das eine Initialisierung kommt.
    outportb(0x20, 0x11);
    outportb(0xA0, 0x11);

    // definiere neue Data-Offset-Vektor für PICs

    outportb(0x21, 0x20);
    outportb(0xA1, 0x28);

    // Initialisierungsvorgang

    outportb(0x21, 0x04);
    outportb(0xA1, 0x02);
}
```

Mit der oben beschriebenen Funktion „idt_set_gate“ kann man jetzt IDT-Einträge Verändern oder Einfügen:

```
void irq_install()
{
    irq_remap();
    idt_set_gate(32, (unsigned) irq0, 0x08, 0x8E);
    idt_set_gate(33, (unsigned) irq1, 0x08, 0x8E);
    idt_set_gate(34, (unsigned) irq2, 0x08, 0x8E);
    idt_set_gate(35, (unsigned) irq3, 0x08, 0x8E);
    idt_set_gate(36, (unsigned) irq4, 0x08, 0x8E);
    idt_set_gate(37, (unsigned) irq5, 0x08, 0x8E);
    idt_set_gate(38, (unsigned) irq6, 0x08, 0x8E);
    idt_set_gate(39, (unsigned) irq7, 0x08, 0x8E);
    idt_set_gate(40, (unsigned) irq8, 0x08, 0x8E);
    idt_set_gate(41, (unsigned) irq9, 0x08, 0x8E);
    idt_set_gate(42, (unsigned) irq10, 0x08, 0x8E);
    idt_set_gate(43, (unsigned) irq11, 0x08, 0x8E);
    idt_set_gate(44, (unsigned) irq12, 0x08, 0x8E);
    idt_set_gate(45, (unsigned) irq13, 0x08, 0x8E);
    idt_set_gate(46, (unsigned) irq14, 0x08, 0x8E);
    idt_set_gate(47, (unsigned) irq15, 0x08, 0x8E);
}
```

Der Codesegment-Selektor ist 0x08. 0x8E setzt sich aus dem Präsenzbits(High nibble) und dem IDT-Eintrag(Low nibble, 0xE = „80386 Interrupt Gate“) zusammen.

IRQ Handler

Die eigentliche Verknüpfung des IRQ mit dem Handler für den Interrupt sowie die Rücksetzung des Master PIC bzw. Master und Slave PIC erfolgen in dieser Funktion:

```
void irq_handler(struct regs* r)
{
    void (*handler)(struct regs* r);

    handler = irq_routines[r->int_no - 32];
    if (handler) { handler(r); }

    if (r->int_no >= 40) { outportb(0xA0, 0x20); }

    outportb(0x20, 0x20);
}
```

System Clock / Programmable Interval Timer (PIT)

Die wichtigsten Ziele eines Betriebssystems ist Raum und Zeit zu beherrschen.

Raum bedeutet, die an die CPU angeschlossene Hardware zu beherrschen und Speicherplatz später auf Langzeit-Datenträgern zu managen.

Zeit bedeutet, man benötigt einen periodischen Taktgeber ("ticks"). Dafür wird ein 1,193182 MHz Quarzoszillator (1/3 des NTSC color burst) als Quelle verwendet.

Es stehen also 3 „Timer“ zu Verfügung:

Timer 0 wird von gängigen OS als System Timer genutzt.

Timer 1 hat keine Bedeutung mehr (historische Bedeutung (RAM refresh)).

Timer 2 kümmert sich um den „Peep“-Sound aus dem PC Lautsprecher

Der Timer zählt von 65535 (16 Bit) auf 0 und beginnt immer wieder von vorne. Dabei wird der IRQ0 zum Prozessor gesendet. Die Frequenz ergibt sich zu $1193182 \text{ Hz} / 65536 = 18,2065 \text{ Hz}$. Es kommt auf diese Art ein Intervall zwischen zwei "ticks" von 1000 Millisekunden / 18,2065 = **54,9254 Millisekunden** zustande. Man erhält also vom Timer 0 ca. 55 mal den IRQ0 pro Sekunde. Diesen periodischen IRQ0 kann ich für zeitgesteuerte Aktionen einsetzen.

Keyboardtreiber von Polling auf IRQ1

Der Keyboardtreiber ist für den IRQ1 noch nicht ausreichend brauchbar. Man muss ein paar Zeilen ändern:

```
void keyboard_init()//warte bis buffer leer ist
{
    while (inportb(0x64)&1)
        inportb(0x60);
};

unsigned int FetchAndAnalyzeScancode()
{
    unsigned int scancode; // variable für den Scancode
    while(TRUE) // warte bis eine taste gedrückt wird
    {
        scancode = inportb(0x60); // port 0x60: holle den scancode

        // drehe das 7 bit, also 1 zu 0 oder 00 zu 1
        unsigned char port_value = inportb(0x61);
        outportb(0x61,port_value | 0x80); // 0->1
        outportb(0x61,port_value &~ 0x80); // 1->0
    }
}
```

Video.c & Keyboard.c wird ausgebaut

Zurzeit ist die Bildschirmausgabe recht eingengt. Man kann nur sagen was man schreiben möchte, welche Linie und mit welcher Farbe. Eigentlich ist k_printf(); ein falscher Name, weil man noch keine Formate ausgeben kann. Ich schreibe die Funktion um, damit man auch einzelne Charakter an einem beliebigen Punkt(x,y) ausgeben kann.

Dazu erstelle ich globale variable für die "Character Attributes", die aktuelle und gespeicherte Bildschirmposition P(x,y) sowie einige Funktionen zum Positionieren des Cursors ein.

Es gibt Funktionen für die Ausgabe von Characters bzw. Strings und das Setzen der Character Attributes. Für diese Funktion benötige ich den Standard-Header <stdarg.h>, der es einer Funktion erlaubt, eine unbestimmte Anzahl an Argumenten zu übernehmen.

Folgende Funktionen habe ich dazugeschrieben:

Settextcolor(vordergrund,hintergrund)	Ändert Textfarbe
Move_cursor_right	Bewegt Cursor eine stelle nach rechts
Move_cursor_left	Bewegt Cursor eine stelle nach links
Move_cursor_home	Setzt Cursor nach ganz links
Move_cursor_end	Setzt Cursor nach ganz rechts
Set_cursor(x,y)	Setzt Cursor nach x,y
Putch(Charakter(Buchstabe))	Für (Tab,\n,\r,backspace,und so)
Puts(text)	Teilt string in eizelne teile auf für Putch
scroll	Scrollt aller Text auf dem Bildschirm
K_printf(nachricht,linie,farbe)	Gibt formate auf Bildschirm aus
Printformat(nachricht)	Wie k_printf unterstützt %u %d %x %s %c
Save_cursor	Speichert aktuelle x,y des Cursor
Restore_Cursor	Setzt Cursor auf Speicherpunkt zurück

Da sich jetzt die Tastatur durch IRQ handle, muss ich auch video.c ausbauen. Jetzt warte ich nicht mehr auf eine Taste, sondern schaue erst, wenn eine Taste gedrückt wurde. Sonderzeichen/Tasten können nun auch benutzt werden (dies bezieht sich auf die Cursor Tasten in keyboard.h):

Taste	Funktion
KINS	-
KDEL	Cursor nach rechts + Backspace
KHOME	Move_cursor_home
KEND	move_cursor_end
KPGUP	
KPGND	
KLEFT	move_cursor_left
KUP	
KDOWN	
KRIGHT	move_cursor_right
Default(a,r,b,4, ,...)	Gibt Taste als Ascii code aus

Systemsfrequenz verändern

Man kann die Frequenz des System Timer selbst in gewissen Grenzen einstellen. Dies erlaubt die nachstehende Funktion:

systemTimer_setFrequency(ULONG freq).

Ich entscheide mich für 100 Hz. Durch das erhalte ich also in regelmäßigen Abständen von jeweils 10 Millisekunden einen "Tick", oder genauer gesagt einen IRQ0. Via Port 0x43 sende ich das Kommando und über 0x40 stelle ich den Teiler ein.

```
static void systemTimer_setFrequency( ULONG freq )
{
    ULONG divisor = 1193180 / freq; //muss 2 Byte sein

    // Sende Kommando
    outportb(0x43, 0x36);

    // Sende Teiler
    outportb(0x40, (UCHAR)( divisor & 0xFF )); // low byte
    outportb(0x40, (UCHAR)( (divisor>>8) & 0xFF )); // high byte
}
```


Abschluss Semesterarbeit, begin freizeit Projekt.

Mein Betriebssystem kann man jetzt schon als kleine Schreibmaschine anschauen. Ich kann zwar noch nichts abspeichern oder laden, aber das kommt. Ich habe mich entschieden an diesem Punkt mit der Semesterarbeit aufzuhören, weil:

- Mir macht es nicht gerade Spass unter Zeitdruck zu Arbeiten, vor allem an etwas, wo Fehler entstehen können wo nur mit einem Hex-editor oder "Hacker"-tool gefunden werden können.
- Fehler zu finden und zu beheben braucht viel Zeit und Geduld, vor allem wenn man in Foren nach seinen Fehlern nachfragen muss.
- Vieles ist auf Englisch und darin bin ich leider nicht gerade der Beste :/ Ich hoffe aber, dass ich dadurch besser werde.
- Ich kann zu Hause viel besser am Projekt arbeiten als in der Schule
- Ich möchte frei arbeiten können und kein Zeitplan einhalten

Darum habe ich mich entschieden an diesem Projekt privat weiter zu Arbeiten. Ich habe noch viele Pläne vor, die mich interessieren an einem Betriebssystem wie z.b. zurzeit USB und Netzwerk. Aber für die muss ich noch viele andere Dinge zuerst erledigen, damit die gehen.

Mein Ziel ist es ein Betriebssystem zu entwickeln, welches ich auch in meinem Alltag brauchen könnte.