CT-Projekt: Mini-OS auf Diskette

Inhalt:

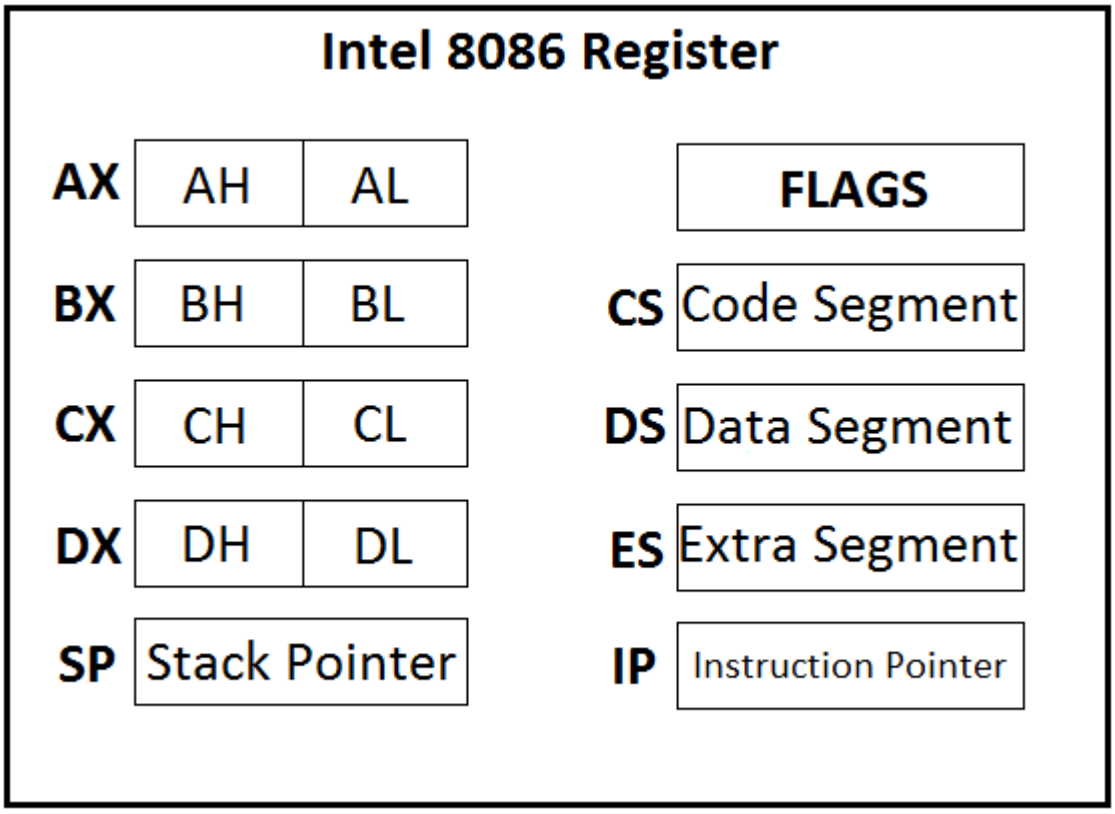
Einleitung

Wessen Traum war es nicht nach den ersten Grundkenntnissen im Programmieren ein eigenes Betriebssystem zu programmieren? Dabei erging es mir wie sicherlich vielen Anderen: Auf der Suche nach Hilfe oder Tutorials im Internet stieß ich auf „unmöglich“, „such dir was einfacheres“ oder endlose Listen mit Anforderungen die ich mir zuerst aneignen sollte. Entmutigt widmete ich mich damals anderen Themen und lies diesen Traum vorerst ruhen…
Doch nun, auf der Suche nach einem CT-Projekt, wagte ich mich erneut an dieses umstrittene Thema und musste feststellen, dass das „Unmögliche“ verblüffend einfach und erreichbar ist. Um Anderen auf dem Weg zur Programmierung des ersten kleinen Betriebssystems Unterstützung und einen Einstieg zu geben möchte ich in den folgenden Seiten das wirklich nötige Grundwissen bis zu einem kleinen Beispiel-OS näher bringen.

Robin Laidig, 17.12.2014

Der Intel 8086 Prozessor

Aller Anfang beginnt mit nicht mit der Programmierung sondern mit ein wenig Grundwissen über die Hardware eines Computers, genauer mit dem Intel 8086, dem ersten serienmäßig produzierten Intel-Prozessor. Selbstverständlich hat sich zur damaligen Technik einiges geändert doch die Kompatibilität ist bis heute gewährleistet da lediglich Funktionen hinzugefügt oder verbessert wurden.
Für uns sind hauptsächlich die unterschiedlichen Registerbezeichnungen wichtig, welche in folgender Grafik zusammengefasst sind:



In **FETT** sind die Registerbezeichnungen dargestellt, welche wir in unserem späteren Quellcode verwenden werden. Jedes Register ist beim 8086 Prozessor 16 Bit groß, die 4 Datenregister AX, BX, CX und DX können aber auch als zwei 8 Bit Register verwendet werden (AX = AH + AL). Es existieren noch ein paar weitere Register, welche für diese Ausarbeitung jedoch nicht benötigt werden.
Erklärung der einzelnen Register:

AX – DX: Datenregister, werden zum Zwischenspeichern vom Programmierer verwendet
SP: Stack Pointer, zeigt auf den aktuellen Eintrag im Stack
FLAGS: Beinhaltet eine Reihe von Bits, sogenannten Flags, die bei bestimmten Ereignissen gesetzt bzw. gelöscht werden.
CS - ES: Diese Register werden zum Adressieren des Hauptspeichers verwendet
IP: Der Instruction Pointer zeigt auf den nächsten auszuführenden Befehl im Programmseicher.

Robin Laidig, 17.12.2014

Auf die genaue Funktion und Anwendung der Register werde ich später am Beispiel des Mini-OS eingehen. Damit können wir auch direkt weiter zur Programmierung gehen!

Grundlegende NASM-Befehle

Um ein Assembler-Programm für annähernd alle heutigen Prozessoren zu schreiben kann man zwischen drei relativ ähnlichen Assemblern wählen:

- **MASM = Microsoft Assembler**
- **TASM = Borland Assembler**
- **NASM = Netwide Assembler**

Wie bereits an der Überschrift zu sehen habe ich mich für den NASM entschieden, da ich bei meinen Recherchen auf sehr viele hilfreiche Quellen und Anlaufstellen gestoßen bin. Um das Mini-OS zu verstehen und selbst einige Assembler-Programme schreiben zu können, sollten folgende Befehle bekannt sein:

Befehl *Operand1* , *Operand 2*	**Beschreibung**
mov ax , bx	Kopiert den Wert von BX in AX.
sub ax, bx	Zieht BX von AX ab und speichert das Ergebnis in AX
add ax, bx	Addiert AX und BX und speichert das Ergebnis in AX
inc ax	AX wird um 1 erhöht
dec ax	AX wird um 1 erniedrigt
cmp ax, bx	Vergleicht AX und BX, FLAGS werden gesetzt
jmp hierhin	Springt zur Markierung „hierhin:“ im Quellcode
jz dorthin	Wenn vorheriger Vergleich = 0, dann Sprung zu „dorthin:“
jnz dahin	Wenn vorheriger Vergleich ≠ 0, dann Sprung zu „dahin:“
jae nirgendwohin	Wenn vorheriger Vergleich ≥ 0, dann Sprung zu „nirgendwohin:“
jbe dochwohin	Wenn vorheriger Vergleich ≤ 0, dann Sprung zu „dochwohin:“

Alle blauen Operanden können auch durch konkrete Zahlen ersetzt werden,
also z.B. **mov** ax, 100.

Besondere Befehle:

int 0x12	BIOS-Interrupt 0x12 wird ausgeführt
call unterprogramm	An der Stelle „unterprogramm:“ wird weitergemacht, bis
ret	Ausgeführt wird, dann wird nach dem call Befehl weitergemacht
ORG 0x7C00	Der nächste Befehl wird an die Stelle 0x7C00 im Programmcode geschrieben.

Mit Hilfe dieser Befehle kann schon annähernd das gesamte Mini-OS programmiert werden, nun müssen wir nur noch unser Medium bootfähig machen, um mit dem Programmieren loslegen zu können!

Die Diskette bootfähig machen

Für mein kleines OS habe ich als Medium eine Diskette gewählt, da diese ähnlich wie ein USB-Stick sehr oft beschrieben werden kann und von jedem neuen und alten PC als Bootmedium unterstützt wird. Dabei muss der Rechner nicht einmal ein Disketten-Laufwerk besitzen, ein USB-Diskettenlauwerk funktionierte bei mir ebenfalls!
Damit das BIOS von der Diskette bootet müssen im Grunde nur zwei Voraussetzungen erfüllt sein:

- Die Diskette muss in der Bootreihenfolge ganz oben stehen/ direkt ausgewählt werden
- Auf der Diskette müssen zwei bestimmte Werte an den richtigen Stellen im ersten Block stehen.

Die Bootreihenfolge kann im BIOS-Setup des Computers eingestellt werden, oder alternativ beim Start des Rechners durch drücken einer angezeigten Taste (meist F9, F11 oder F12) direkt für den aktuellen Bootvorgang ausgewählt werden.
Ist diese Voraussetzung erfüllt, können wir uns um die benötigten Werte kümmern, das geht überraschend einfach mit ein paar Assembler-Befehlen:

```
ORG 0x7C00      Der nächste Befehl wird an die Stelle 0x7C00 auf der Diskette geschrieben
jmp start       Wir springen zum „start:“ des Programms

start:
        Das eigentliche Programm…
        Ende des Programms

times 512-($-$$)-2 db 0     Ab dem Ende des Programms den restlichen Platz mit 0 füllen
                            und am Schluss zwei Byte freilassen
dw 0xAA55                   Den Wert 0xAA55 auf die Diskette schreiben
```

Wenn dieser Quellcode (ohne die *Kommentare*) kopiert wird, ist die Diskette bootfähig und wird den Quellcode nach der „start:“-Markierung ausführen.
Was passiert genau?
Zunächst untersucht das BIOS die Diskette und schaut sich die letzten 2 Bytes an. Steht an dieser Stelle der Wert 0xAA55 so wird die Diskette als bootfähig eingestuft. Nun wird der erste Befehl an der Stelle 0x7C00 ausgeführt. Da wir hier unseren jmp-Befehl platziert haben, springt das Programm zur „start:“-Markierung und wir können beliebige weitere Befehle ausführen. Sollte das Programm beim letzten Byte ankommen, so startet der PC neu.

Robin Laidig, 17.12.2014

Um direkt einen Neustart auszulösen, also das Programm zu beenden, kann einfach ein Sprung zur Stelle FFFF ausgeführt werden: jmp 0xFFFF.

Somit ist alles geklärt, bis auf den times-Befehl. Dazu muss man wissen, dass der Assembler ein Programm ist, welches die für uns logischen Befehle in 0en und 1en umwandelt, dem sogenannten Opcode. Hierbei kann man aber auch dem Assembler Befehle zukommen lassen, welche beim Assemblieren (Quellcode -> Opcode) berücksichtigt werden, jedoch nicht im Programm als Befehle enthalten sind. Beispiel dafür ist der ORG-Befehl, er sagt lediglich dem Assembler, dass der nächste Befehl an die angegebene Position geschrieben werden soll.
Der **times-Befehl ist eine weitere Anweisung an den Assembler die eine Wiederholung** ausdrückt, genauso wie db ein Byte an die aktuelle Stelle schreibt und dw 2 Byte (ein WORD) an die aktuelle Stelle schreibt.
Somit bewirkt „times 10 db 0“ zehn aufeinanderfolgende Byte mit dem Wert 0 und „times 20 db 123“ zwanzig aufeinanderfolgende Byte mit dem Wert 123.
Das $-Zeichen wird vom Assembler durch die aktuelle Position im Programm ersetzt und das $$-Zeichen durch den Anfang des Programms. ($-$$) beinhaltet also die Länge des Programms bis zur aktuellen Position.

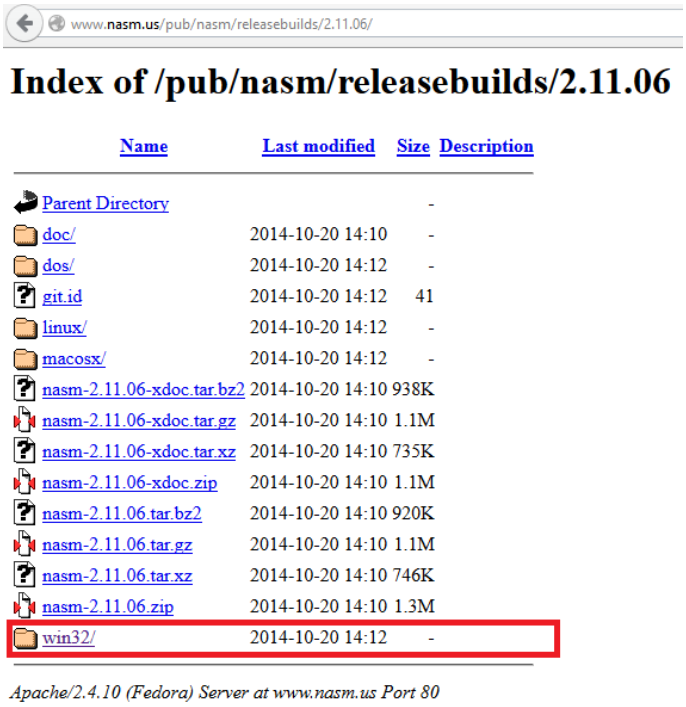
times 512 – ($-$$) – 2 db 0

beschreibt also den restlichen Platz des Blocks (welcher 512 Byte groß ist) mit Nullen, mit Ausnahme der letzten zwei Byte.

Den Quellcode haben wir nun, aber wie bekommt man das Ganze auf eine Diskette?

1. **NASM Installieren**

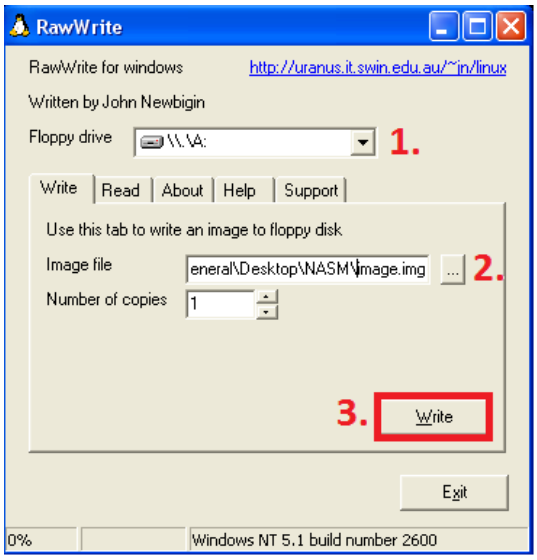
Zu aller Erst benötigen wir natürlich den NASM-Assembler, der den Quellcode für uns in Opcode umwandelt und uns dabei auf Fehler hinweist. Diesen kann man ganz einfach unter http://www.nasm.us unter der Rubrik „Downloads“ runterladen, für Windows befindet sich der Installer im „win32“ Ordner.
Nach der Installation navigieren wir in den NASM-Order, den wir bei der Installation festgelegt haben und erstellen dort eine neue Textdatei. In diese Textdatei fügen wir nun unseren Quellcode ein speichern sie anschließend als „quellcode.asm“. Um diese Datei nun zu kompilieren können



wir entweder direkt die Verknüpfung auf dem Desktop verwenden, oder über die Windows CMD in den Ordner manövrieren. Dann geben wir den Befehl
nasm –f bin –o „C:\PfadzurDatei\ausgabe.bin“ „C:\PafdzurDatei\quellcode.asm“
ein und erhalten eine kompilierte bin-Datei. Um diese nun auf die Diskette zu schreiben müssen wir sie lediglich zu einer IMG-Datei zusammenfassen, dies geht über folgenden Befehl:
copy /b „C:\PfadZurDatei\ausgabe.bin“ „C:\PfadZurDatei\image.img“
Damit haben wir eine img-Datei, die wir nun auf die Diskette schreiben können.

2. Datei mit RawWrite auf Diskette schreiben

RawWrite ist ein kostenloses, sehr nützliches Programm, das zum Beispiel bei Heise.de heruntergeladen werden kann: http://www.heise.de/download/rawwrite.html
Wir müssen lediglich unser Diskettenlaufwerk und die Image-Datei auswählen, mit dem Klick auf „write“ wird die Datei 1:1 auf die Diskette geschrieben. Nun ist die Diskette für Windows nicht mehr lesbar und wir können sie beim Booten testen!
Damit sind alle Voraussetzungen erfüllt, dass das Mini-OS getestet und verstanden werden kann..



Mini-OS

Hier habe ich meinen kommentierten Quellcode der gerne als Anstoß/Grundgerüst für eigene Projekte verwendet werden kann!

```
ORG 0x7C00
jmp 0x0000:start

start:
        cli                     ;Interruptverbot

        mov ax,0x9000           ;Register AX = 0x9000
        mov ss,ax               ;SS = Stackbeginn
        mov sp,0x9000           ;SP = Stackpointer = 0x9000 + 0x0000

        sti                     ;Interruptfreigabe

        mov ah, 0x00
        mov al, 0x13
        int 0x10                ;Video Mode 320x200 @ 256 Farben
```

```
        mov ax, 40960
        mov ds, ax              ;DS 40960 = Video-Buffer


        mov ax, 1               ;ax = Farbe, bx = Pixel. Oben Links = 0, Unten Rechts = 319*200
        mov bx, 320*28
line:                           ;Zeichnet eine Linie mit allen 256 Farben, Höhe = 28 Pixel von oben
        mov [bx], ax            ;[bx] schreibt an die Adresse bx im
                                ;Hauptspeicherbereich DS
        inc bx
        inc ax
        cmp bx, 320*28+320      ;BX – 320*28+320 (BX hat rechten Rand erreicht?)
        jnz line                ;Sprung wenn Ergebnis nicht 0

        mov bx, 29*320          ;BX = Pixel-Nummer.
        mov ax, 32              ;Farbe 32 = Blau
        mov [bx], ax            ;Unterhalb der Linie ist ein 2x2 großes Pixel zu sehen.
        add bx, 320             ;Dieses wird zunächst Oben Links, dann UL, dann UR, dann OR gezeichnet
        mov [bx], ax
        add bx, 1
        mov [bx], ax
        sub bx, 320
        mov [bx], ax
        sub bx, 1


        jmp kernel              ;Sprung zur Endlosschleife „kernel“

kernel:
                                ;Das Hauptprogramm kernel
        call getkey             ;Unterprogramm getkey
        call auswertekey        ;Unterprogramm auswerekey
        jmp kernel



getkey:
        mov ah, 0               ;Funktion 0: Warten Auf Tastendruck, AL = ASCII-Code der Taste
        int 0x16                ;BIOS Interrupt 0x16 = Keyboard-Aktionen. Genaue Funktion in ah gespeichert
        ret                     ;Unterprogramm fertig, zurück zum Hauptprogramm

auswertekey:
        cmp al, 13              ;Enter-Nummer abziehen
        jz key_Enter            ;Wenn al – 13 = 0, dann Sprung zu key_Enter
        cmp cx,1                ;CX speichert den Modus, 1 = Textausgabe, 2 = Pixel steuern
        jz key_mode_1           ;wenn CX = 1, dann Sprung zu key_mode_1
        cmp al, 0x1B            ;ESC-Nummer abziehen
        jz key_ESC              ;Jump if Zero zu key_ESC
        cmp al, 119             ;119 = w
        jz key_w
        cmp al, 97              ; 97 = a
        jz key_a
        cmp al, 115             ;115 = s
        jz key_s
        cmp al, 100             ;100 = d
        jz key_d
```

```
        ret

key_mode_1:
        mov dx, bx              ;bx in dx sichern
        mov ah, 0x0E            ;AH = 0x0E = Funktion zum Zeichen ausgeben, welches in AL gespeichert ist
        mov bx, 40              ;Farbe 40 = rot
        int 0x10                ;Interrupt für Zeichenausgabe
        mov bx, dx              ;bx wiederherstellen
        ret

key_ESC:
        jmp 0xFFFF              ;reboot
key_Enter:
        cmp cx, 1               ;CX - 1
        jz adden                ;Wenn 0, dann Sprung zu „adden“
        mov cx, 1               ;Wenn nicht weggesprungen, dann CX = 1
        ret
adden:
        mov cx, 2               ;CX = 2
        ret
key_w:
        cmp bx, 320*30-1        ;Obere Barriere
        jbe ende_auswertekey
        mov ax, 0
        mov [bx], ax            ;OL löschen
        add bx, 320
        mov [bx], ax            ;OR löschen
        add bx, 1
        mov [bx], ax            ;UR löschen
        sub bx, 320
        mov [bx], ax            ;UL löschen
        sub bx, 1               ;B wieder Wert von OL (normal)

        mov ax, 32              ;Farbe ändern
        sub bx, 320             ;B-=320

        mov [bx], ax            ;OL schreiben
        add bx, 320
        mov [bx], ax            ;OR schreiben
        add bx, 1
        mov [bx], ax            ;UR schreiben
        sub bx, 320
        mov [bx], ax            ;UL schreiben [] = An Adresse im Speicher wird geschrieben. Speicherbereich = DS =
                                ;Videobuffer
        sub bx, 1
        ret

key_a:
        cmp bx, 320*29          ;Oben Links rausfahren
        jz ende_auswertekey
        mov ax, 0
        mov [bx], ax            ;OL löschen
        add bx, 320
        mov [bx], ax            ;OR löschen
        add bx, 1
        mov [bx], ax            ;UR löschen
```

```
        sub bx, 320
        mov [bx], ax            ;UL löschen
        sub bx, 1               ;B wieder normal

        mov ax, 32              ;Farbe ändern
        sub bx, 1               ;B--

        mov [bx], ax            ;OL schreiben
        add bx, 320
        mov [bx], ax            ;OR schreiben
        add bx, 1
        mov [bx], ax            ;UR schreiben
        sub bx, 320
        mov [bx], ax            ;UL schreiben []
        sub bx, 1
        ret
key_s:
        cmp bx, 320*198         ;Untere Barriere
        jae ende_auswertekey
        mov ax, 0
        mov [bx], ax
        add bx, 320
        mov [bx], ax
        add bx, 1
        mov [bx], ax
        sub bx, 320
        mov [bx], ax
        sub bx, 1               ;B wieder normal

        mov ax, 32              ;Farbe ändern
        add bx, 320             ;B+=320

        mov [bx], ax
        add bx, 320
        mov [bx], ax
        add bx, 1
        mov [bx], ax
        sub bx, 320
        mov [bx], ax
        sub bx, 1
        ret
key_d:
        cmp bx, 320*199-2       ;Unten Rechts rausfahren
        jz ende_auswertekey
        mov ax, 0
        mov [bx], ax
        add bx, 320
        mov [bx], ax
        add bx, 1
        mov [bx], ax
        sub bx, 320
        mov [bx], ax
        sub bx, 1               ;B wieder normal

        mov ax, 32              ;Farbe ändern
        add bx, 1               ;B++
```

```
        mov [bx], ax
        add bx, 320
        mov [bx], ax
        add bx, 1
        mov [bx], ax
        sub bx, 320
        mov [bx], ax
        sub bx, 1
        ret
ende_auswertekey:
        ret


;Nur Assembler Anweisungen um Block zu füllen!

times 512-($-$$)-2 db 0     ;times = Wiederholung des selben Assemblerbefehls,$ = Aktuelle Position, $$ =
                            ;Startposition des Codes. 512(1 Sektor) - unser Programmcode wird mit 0en
                            ;aufgefüllt und am Ende 2 Byte Platz gelassen (-2).
dw 0xAA55                   ;Benötigter Wert (0xAA55) zur Identifikation als bootfähiges Medium wird ans Ende
                            ;des 1. Sektors geschrieben.
```

Ich hoffe mit den Kommentaren und den vorhergegangenen Erklärungen ist der Quellcode größtenteils verständlich.
Nachdem das BIOS von der Diskette gebootet hat, wird zunächst ein Interrupt-Verbot verhängt, damit der Stack, eine Art stapelförmiger Zwischenspeicher, ohne Unterbrechung angelegt werden kann. Dann wird der Video-Modus auf 320x200 Pixel gesetzt und direkt darauf eine Linie in Zeile 28 mit allen 256 Farben gezeichnet.
Zuletzt wird in die Obere Linke Ecke, unterhalb der Trennlinie, ein 2x2 Pixel in blau angezeigt und die Endlosschleife, welche die Tastatur abfragt, beginnt.
Nun kann über die W-A-S-D – Tasten das blaue Pixel im unteren Bereich beliebig bewegt werden, auch ein herausfahren auf der Linken und Rechten Seite ist möglich. Dabei verringert sich Links die Höhe um 1 während sie rechts um 1 ansteigt, dies liegt an der 1-Dimensionalität des Video-Buffers.
Über die Eingabe-Taste kann zwischen zwei Modi gewechselt werden, im zweiten Modus werden im

Robin Laidig, 17.12.2014

oberen Bereich die gedrückten Tasten ausgegeben. Dabei wird das standardmäßige englische Tastaturlayout verwendet.
Mit der ESC-Taste wird die Endlosschleife verlassen und das Programm beendet, ein Neustart des Rechners ist die Folge!

Ich hoffe ich konnte durch meine Ausarbeitung die Grundzüge zur Programmierung eines eigenen kleinen Betriebssystems näher bringen, um selbst zu experimentieren und zu verstehen darf mein Quellcode gerne kopiert und verändert werden!

Robin Laidig, 17.12.2014