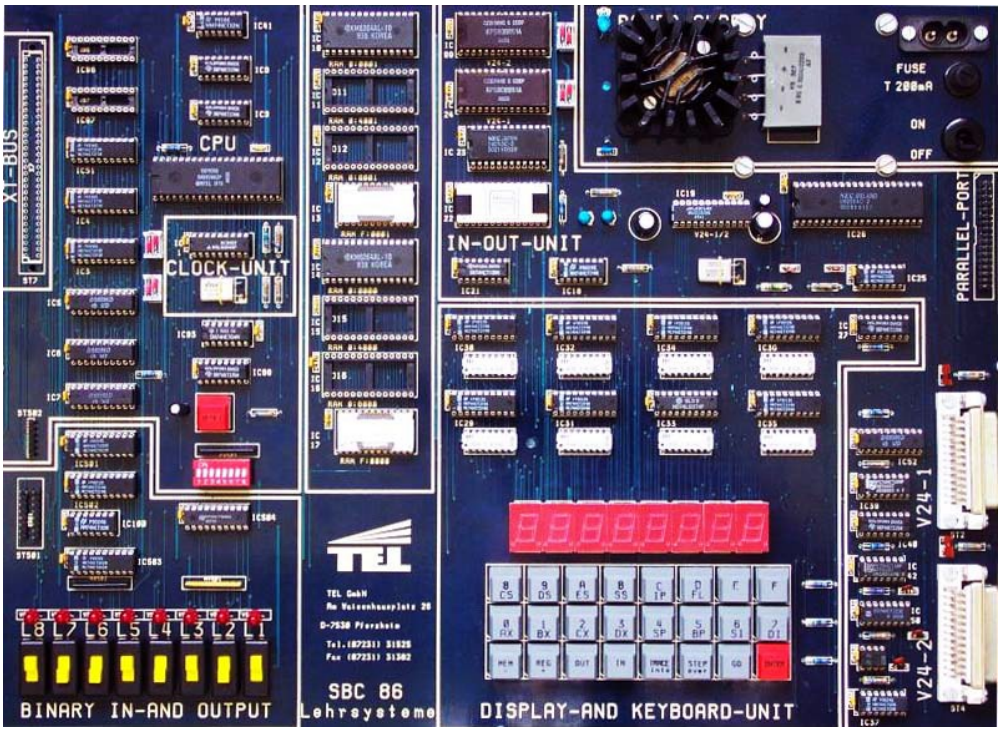
Emulator des Singleboardcomputer SBC 86

Robert Dinse
Fred Brodmüller
Christan Steineck
Jörg Müller-Hipper

Inhaltsverzeichnis

1 Einführung

Der SBC86 ist ein Singleboardcomputer auf Basis der INTEL 8086 CPU, mit 5 MHz und 16 Bit Registern. Er besitzt binäre Ein- und Ausgabe Einheiten, 8 LEDs und 8 Schalter. Für Ein- und Ausgaben stehen acht 7- Segment-Anzeigen und eine kleine Tastatur zur Verfügung.



Der SBC86 wird für Assembler-Lehrveranstaltungen genutzt, da er noch eine geringe Anzahl an Befehlen besitzt, gegenüber den heutigen CPUs. Wir wollten mit dem SBC86-Emulator die Möglichkeit geben, die Kenntnisse in der Assembler-Programmierung auch zu Hause, ohne den SBC, weiter zu entwickeln oder zu vertiefen.

Der SBC86 Emulator ist ein vollwertiger INTEL-8086 Emulator. Er arbeitet annährend in der gleichen Geschwindigkeit, also ungefähr 5 MHz. Der RAM beträgt 64 KByte. Man kann durch die vorhandene Geräte-Schnittstelle zusätzliche Geräte für den Emulator bereitstellen. Es können ausschließlich Maschinenkode-Dateien im 16Bit Rohformat (COM), keine exe oder elf Binärdateien verarbeitet werden. Entsprechende Dateien lassen sich z.B. mit NASM bzw. TASM (Model Tiny) erstellen.

2 Betriebsanleitung des SBC86 Emulators

2.1 Systemvoraussetzungen

- x86 basierender PC mit min. 166MHz

Unter Linux

-gcc Version 2.95 oder höher
-funktionierende X-Windows Installation mit Windowmanager (http://www.xfree.org/)
-GTK-2.0 oder höher (http://www.gtk.org)
-nasm um die Beispiel ASM Dateien zu Assemblieren
(zu finden unter http://nasm.sourceforge.net/)
-für das Beep-Plugin wird der Shellbefehl „beep“ (http://www.johnath.com/beep/) benötigt - dieser muß die SUID haben

Unter Win32

-Windows 9x/2000/XP

Für die Compilierung wird CygWin (http://www.cygwin.com/) benötigt.
Dafür werden folgende Bibliotheken verwendet.

-GTK-2.0 für CygWin oder höher. Um GTK unter CygWin zu installieren muss in der setup.exe von CygWin in die Eingabefläche “User URL“ die Adresse “http://web.sfc.keio.ac.jp/~s01397ms/cygwin/“ eingetragen werden. Von den dort erhältlichen Paketen muss Libs/gtk2-win32 und Libs/atk installiert werden.
(http://web.sfc.keio.ac.jp/~s01397ms/cygwin/index.html.en)
-ncurses

-nasm um die Beispiel ASM-Dateien zu Assemblieren
(zu finden unter http://nasm.sourceforge.net/)

2.1.1 Installation

Linux

-in einer Konsole in das Verzeichnis i8086emu/src wechseln
-“make“ aufrufen (GTK2 Version wird erstellt)
-um das i8086sic-Plugin (serielles Interface) zu nutzen muss die emu.cfg angepasst werden
-zum Starten des Emulators i8086gui aufrufen

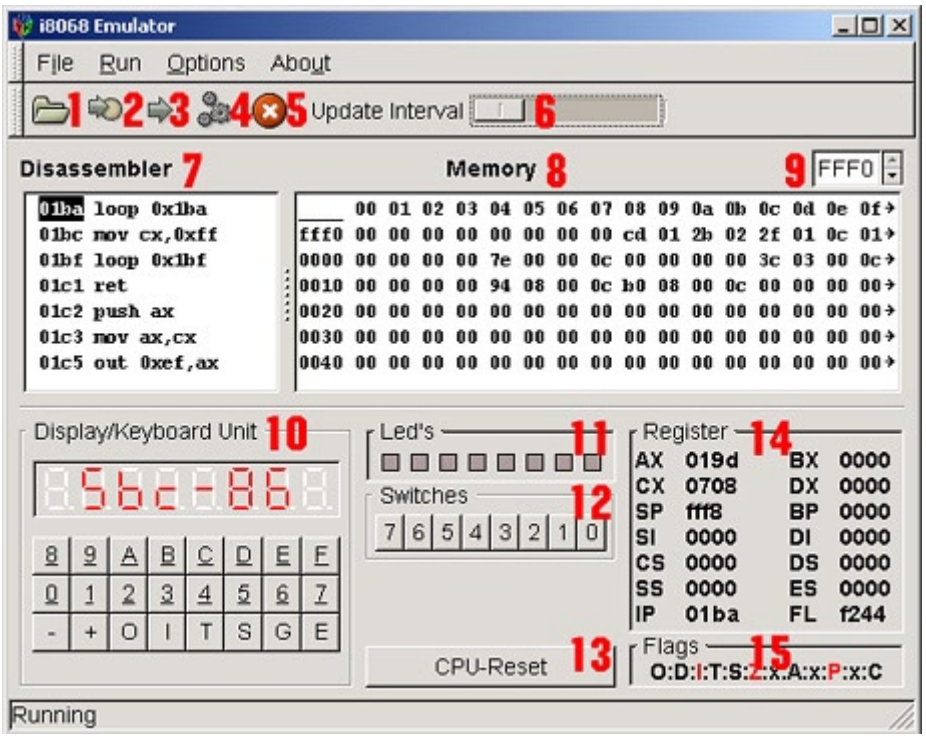
Windows

-i8086setup.exe doppelt klicken und den Bildschirmanweisungen folgen
-zum Starten des Emulators Start->Programme->i8086emu->i8086emu anklicken

2.2 i8086GUI

i8086gui ist eine grafische Oberfläche für den SBC86 Emulator. Sie ist mit GTK-2.0 erstellt und damit auf verschiedene Plattformen portierbar.

2.2.1 Oberfläche



1: Open, öffnet eine Datei im COM Format.

2: Trace, Einzelschritt, Es wird nur der nächste Befehl des Programms ausgeführt.

3: Step Over, Im Unterschied zu Trace werden hier Unterprogramme und bedingte Programmschleifen insgesamt abgearbeitet.

4: Run, Das Programm wird an der aktuellen Position gestartet. Eine gezielte Unterbrechung ist nur mittels eines Breakpoints oder über Stop möglich. Nur in diesem Modus kommt die Einstellung des Update Intervall(6) zum tragen.

5: Stop, beendet den Run Modus.

6: Update Interval, Einstellung wie oft die Register und Flags im Run Modus aktualisiert werden.

7: Dis-Assembler, Anzeige der Befehle, die abgearbeitet werden, der markiere Bereich ist der nächste Befehl. Breakpoints erscheinen rot Markiert.

8: MEM-Viewer, Anzeige des Speichers, markierter Bereich ist die Speicheradresse des nächsten Befehls der abgearbeitet wird.

9: Display Memory, Eingabe der Adresse, ab der der Speicherinhalt angezeigt werden soll.

10: Segmentanzeige- und Eingabefeld, Auf die acht 7-Segment-Anzeigen kann über die Ports 90h-9eh zugegriffen werden. Dabei ist jedes Segment mit einer der acht Ausgangsleitungen des entsprechenden Ports verbunden. Die Tastaturmatrix kann über den Port 80h abgefragt werden.

11: LED Anzeige, Auf die acht LEDs kann über den Port 00H zugegriffen werden.

12: Switches, Den Zustand der Schalter kann man ebenfalls über den Port 00H abfragen.

13: CPU-Reset, Neustart der CPU, das bedeutet, dass die Abarbeitung des laufenden Programms abgebrochen wird. Danach wird an das Ende des Speichers gesprungen, an der eine Routine des ROM liegt.

14: Register, Anzeige der Register mit ihren Inhalten.

15: Flags, Anzeige der Flags, im gesetzten Zustand sind sie rot eingefärbt.

2.2.2 Das Menü

File- Menü: -Open, Öffnet ein Assembler File im COM Format.
-Exit, beendet das Programm.

Run- Menü: -Trace, Einzelschritt, Es wird nur der nächste Befehl des Programms ausgeführt.
-Step Over, Im Unterschied zu Trace werden hier Unterprogramme und bedingte Programmschleifen insgesamt abgearbeitet.
-Run, Das Programm wird an der aktuellen Position gestartet. Eine gezielte Unterbrechung ist nur mittels eines Breakpoints oder über Stop möglich. Nur in diesem Modus kommt die Einstellung des Update Intervall zum tragen.
-Stop, beendet den RUN- Modus.

Option- Menü: -Change Register, ändern eines Register Inhaltes Gewünschtes Register auswählen und dann den Wert in hexadezimalen Zahlen eingeben.
-Change Memory, Adresse in hexadezimaler Form eingeben dann gewünschten Wert ebenfalls in hexadezimaler Form

2.2.3 Shortcut's

CTRL + O	- Open	CTRL + M	- Change Memory
CTRL + Q	- Quit	CTRL + B	- Set/Del Breakpoint
F7	- Trace	CTRL + R	- Change Register
F8	- Step Over	Alt + 0-9	- Zahlen des Keyboards
F9	- Run	Alt + A-F	- Buchstaben des Keyboards
Esc	- Stop		

2.3 i8086text

i8086text ist der reine Emulator mit einer kleinen GUI im Terminal Fenster. Wird in der i8086text-Version das ROM-File aktiviert (siehe Kapitel 2.4), kann keine Programmdatei ausgeführt werden. Soll eine eigenes Programm geladen werden, welches Funktionen des ROMs verwendet, muss der Core-Dump aktiviert werden.

Kommandozeilen-Parameter

- [Options] Dateiname - Datei im COM Format
- [-c] - CodeViewer deaktivieren
- [-o] xxxxh - Startadresse des Programms
- [-r] xxxxh file - Startadresse des ROM-Files
- [-d] file - DUMP-File

Funktionstasten im Emulator

q -Quit, Emulator schließen

r -Change Register, ändern eines Register Inhaltes. Zahl für das gewünschte Register und dann den Wert als dezimale Zahl eingeben.

w -Change Memory, Adresse in hexadezimaler Form eingeben, dann denn gewünschten Wert ebenfalls in hexadezimaler Form

m -Display Memory, Eingabe der Adresse (in Hex), ab der der Speicherinhalt angezeigt werden soll.

F1 -Hilfe, Listet alle Debuggerbefehle einschließlich ihrer Syntax auf

F2 -Set Breakpoint, Haltepunkt setzen. Es kann eine Adresse angegeben werden, bei der die Programmabarbeitung(RUN) unterbrochen werden soll.

F3 -Write Core, schreibt den kompletten Speicher in die Core-Dump-Datei.

F7 | T -Trace, Einzelschritt. Das gesamte Programm wird Befehl für Befehl abgearbeitet

F8 | N -Step Over / Next, Im Unterschied zu Trace werden hier Unterprogramme und bedingte Programmschleifen insgesamt abgearbeitet.

F9 -Run, Das Programm wird an der aktuellen Position gestartet.

F12 -Reset, Neustart der CPU, das bedeutet, dass die Abarbeitung des laufenden Programms abgebrochen wird und an das ende des Speichers gesprungen wird, an der eine Routine des ROM liegt.

0-7 -Mit den Tasten 0-7 kann man den Zustand der Schalter beeinflussen.

2.4 Konfigurations-Datei

In der Konfigurations-Datei (emu.cfg) des Emulators können die Grundeinstellungen festgelegt werden. Mögliche Werte für die Konfigurationseinträge können als Zeichenkette., Hexadezimaler-, Dezimaler- und Wahrheitswert eingetragen werden. Mit # beginnt ein Kommentar.

Es wird empfohlen die Standard-Einstellungen beizubehalten.

2.4.1 Grundeinstellungen

PROGSTARTADR Adresse im Hex-Format, an die der Emulator die Programme lädt.

ROMFILE Falls diese Option verwendet wird, lädt der Emulator beim Start die angegebene Datei an ROMSTARTADR in den Speicher und beginnt die Ausführung an dieser Adresse. Mit der dem Emulator beiliegenden ROM-Version ist eine andere Adresse als C000h nicht nutzbar.

ROMSTARTADR Adresse im Hex-Format, an die das ROM geladen wird (nur sinnvoll in Kombination mit ROMFILE).

COREDUMP Ermöglicht das Laden eines Speicherabbildes (an Adresse 0h) beim Starten des Emulators. Mit der Textversion des Emulators ist es möglich ein solches zu erstellen.

Achtung: Die Optionen ROMFILE und COREDUMP schliessen sich gegenseitig aus.

2.4.2 Geräte-Einstellungen

Der Emulator unterstützt dynamisch ladbare Geräte. Diese werden wie folgt konfiguriert.

DEVICECOUNT Anzahl der Geräte die eingebunden werden
DEVICE*n* Voller Pfad und Name zur Datei des Geräts n. Das erste Gerät beginnt mit der Nummer 0.

2.5 Grundlegende Arbeitsweise des SBC86 Emulators

In diesem Teil der Dokumentation wird kurz beschrieben, was ab dem Start bis zum Beenden des Emulators geschieht. Als erstes wird der Prozessor initialisiert, hierbei werden vom Prozessor mehrere Tätigkeiten ausgeführt (siehe Kapitel 4.2), z.B. die Ports und die Kommandostruktur zu initialisieren. Danach wird die Rom- oder die Core-Dump- Datei in den Speicher geladen. Wenn die Rom-Datei geladen wurde wird sie zunächst ausgeführt, da sie z.B. die Interrupt-Vektortabelle anlegt. Nun werden die Devices geladen und eingebunden. Ab diesem Punkt läuft der Emulator schon und es kann ein Programm geöffnet und ausgeführt werden. Bei dem Schließvorgang wird der Prozessor geschlossen und alle Ressourcen wieder freigegeben.

i8086arithmetic.c, *i8086control.c*, *i8086controltrans.c*, *i8086datatrans.c*, *i8086logic.c*, *i8086strings.c*
Enthalten die Emulationsfunktionen für die Arithmetik -, Kontroll-, Kontrolltransfer-, Datentransfer-, Logik- und Stringmanipulationsoperationen des 8086 Prozessors.

i8086config.c
Funktionen zum Schreiben und Lesen der Konfiguration in bzw. aus einer Textdatei (emu.cfg).

i8086devices.c
Pluginschnittstelle für die dynamisch ladbaren Geräte (siehe Kapitel 7)

i8086text.c
Bei der Entwicklung des Emulators wurde eine ncurses Oberfläche benutzt. Diese ist in der Datei i8086text.c zu finden. Der Code wurde hier jedoch kaum dokumentiert, da es sich nicht um die offizielle Oberfläche des Emulators handelt, sondern nur um eine Testumgebung.

i8086emufuncs.c
Die möglichen Befehle der CPU werden in der Datei i8086emufuncs.c mit ihrem Opcode und der zugehörigen Emulatorfunktion „registriert“. Weiterhin einige Hilfsfunktionen, unter anderem zum Zugriff auf Speicher und Ports.

i8086error.c
Alle Fehlercodes und Funktionen zum Schreiben in das Logfile.

*i8086gui**
Die GTK2-Oberfläche ist auf die Dateien i8086gui.c, i8086gui_emufuncs.c, i8086gui_error.c, i8086gui_paint.c und i8086gui_util.c verteilt. Diese stellt die offizielle Oberfläche des Emulators dar.

i8086messages.c
Die Funktionen und Datenstrukturen zur Interprozesskommunikation via Nachrichten sind in der Datei i8086messages.c zu finden.

i8086proc.c
Enthält Registerzugriffs - und unmittelbar den Prozessor betreffende Funktionen sowie die Emulatorhauptschleife

i8086util.c
Einige Hilfsfunktionen.

i8086wrapper.c
Doppelte Opcodes einiger Assembler Befehle werden in der i8086wrapper.c unterschieden.

i8086pit.c
Implementierung des programmierbaren Interrupt Timers als Plugin

i8086pic.c
Implementierung des programmierbaren Interrupt Kontrollers als Plugin
Diese beiden Geräte benutzen die Plugin API des Emulators.

Die Prototypen und Konstantendefinitionen sind in den zugehörigen Headerdateien zu finden.

Im Verzeichnis disasm befindet sich der ndisasm source code, den wir für den Emulator als Disassembler benutzt haben.

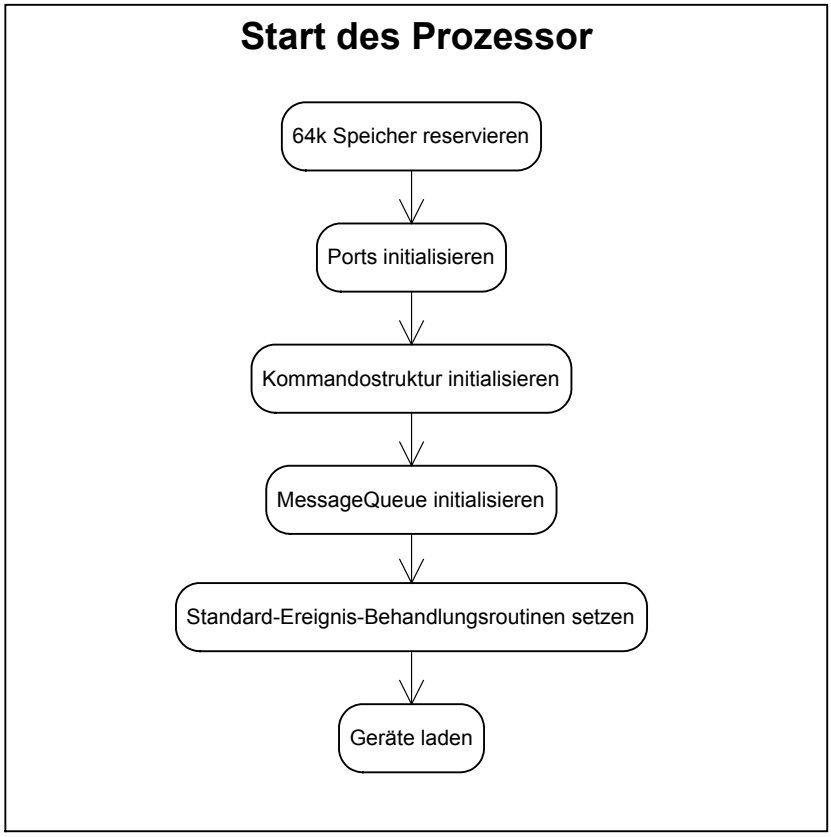
4 Prozessor

4.1 Einführung

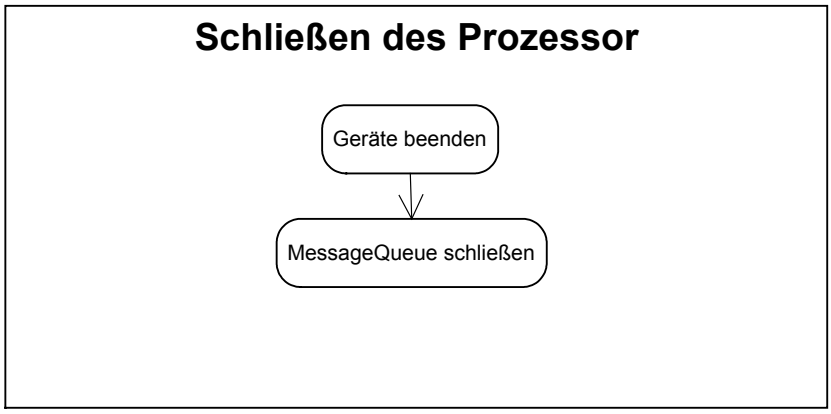
Der Prozessor des i8086emu besteht aus den Grundfunktionen für die Befehlsverarbeitung, den Registern und dem Speicher. Der aktuelle Zustand des Prozessors wird in einer Variable vom Typ *i8086core* gespeichert. Bei jedem Takt wird ein Befehl aus dem Speicher geladen und atomar ausgeführt. Danach wird die Nachrichtenliste (siehe Kapitel 6) verarbeitet. Alle *SYNC_CLOCKS* Takte wird die benötigte Zeit gemessen und daraus berechnet wie lange der Prozessor pausiert werden muss, um ihn an die Geschwindigkeit des reellen 8086 anzupassen.

4.2 Initialisierung und Beendigung

Mit der Funktion *i8086init* wird er Prozessor initialisiert. Dabei werden alle benötigten Ressourcen reserviert und vordefinierte Standardwerte eingenommen.



Mit der Funktion *i8086close* werden alle vom Prozessor benötigten Ressourcen freigegeben.



4.3 Prozessor-Funktionen

4.3.1 Datenstrukturen

#include "i8086proc.h"

Der *i8086core* speichert den aktuellen Zustand des Prozessors.

```
typedef struct i8086core
{
  i8086Register reg;          /* Register */
  int flags;                  /* Flags */
  i8086IOPorts ports;         /*IO Ports */
  unsigned short pc;          /* Programmcounter */
  unsigned char *mem;         /* Speicher(64k) */
  i8086MiscData miscData;  /* Zusaetzliche Daten fuer den Programmablauf */
}i8086core;
```

i8086Parameter stellt den Parameter eines Prozessorbefehls dar.

```
typedef union i8086Parameter
{
  long long all;
  int i[2];
  unsigned short w[4]; // 0=l
  unsigned char b[8];  // 0=l
}i8086Parameter;
```

Die *i8086command* repräsentiert einen Befehl des 8086.

```
typedef struct i8086command
{
  char name[i8086_OPC_NAME_LEN]; /* Befehlsname */
  unsigned char opcode;   /* Befehl */
  unsigned char size;     /* Groesse des Befehls mit Parametern in Byte */
  unsigned char hasMod;  /* 1=Mod-Tag existiert, 0=kein Mod-Tag vorhanden */
  unsigned char clocks;   /* Takte des Befehls   */
  void (*func)(i8086core *core, unsigned char opcode, i8086Parameter para,
               i8086Parameter data); /* Emulatorfunktion  */
}i8086command;
```

Für die Aufbewahrung eines Registers steht der Datentyp *i8086SingleRegister* zur Verfügung.

```
typedef union i8086SingleRegister
{
  unsigned short x;
  unsigned char b[2]; /* 0=lByte, 1=hByte */
}i8086SingleRegister;
```

In der *i8086proc.h* sind die folgenden Variablen deklariert.

```
i8086core *core;              /* aktueller Zustand des Prozessors   */
i8086command **commands;      /* Zeiger zu den verfügbaren          */
                              /* Befehlsfunktionen                  */
```

4.3.2 Registerfunktionen

*void i8086GetRegisterRef(i8086SingleRegister *buf, i8086core *core, i8086W w, int reg)*
*void i8086SetRegister(i8086core *core, int reg, i8086W w, unsigned short value)*
*unsigned short i8086GetSegRegister_fast(i8086core *core, int reg, int convert)*
*void i8086SetSegRegister(i8086core *core, int reg, unsigned short value)*

Um auf eines der Register (AX,BX,CX,DX,SP,BP,SI,DI) des i8086 zuzugreifen gibt es die Funktion i8086GetRegisterRef bzw. i8086SetRegister. Die Funktionsweise wird an folgendem Beispiel deutlich:

Beispiel:

Kopieren eines 16Bit Wertes von BX nach DX

```
#include "i8086proc.h"

i8086SingleRegister bx;
unsigned short value;

i8086GetRegisterRef(&bx, core, 1, i8086_REG_BX);
value=bx.x;
i8086SetRegister(core, i8086_REG_DX, 1, value);
```

bx.x - 16Bit des Registers
bx.b[0] - untere 8Bit des Registers (BL)
bx.b[1] - obere 8Bit des Registers (BH)
w gibt die Operationsgröße an (*w*=0 - 8Bit, *w*=1 - 16 Bit)

Um auf die Segmentregister (CS, DS, ES ,SS) des i8086 zuzugreifen gibt es die beiden Funktionen *i8086GetSegRegister_fast* und *i8086SetSegRegister.*

Beispiel:

Kopieren des 16 Bit Wertes von SS nach CS

```
unsigned short ssvalue;

ssvalue=i8086GetSegRegister_fast(core, i8086_REG_SS, 0);
i8086SetSegRegister(core, i8086_REG_CS, ssvalue);
```

Der Parameter *convert* bestimmt den Wert der von *i8086GetSegRegister_fast* zurück gegeben wird. Ist *convert* = 1 so wird der Wert des Segmentregisters mit 0x10 multipliziert - sonst normal ausgegeben. Die Konstanten für die Register sind in *i8086proc.h* definiert.

4.3.3 Ressourcen-Funktionen

void i8086init()

Reserviert alle benötigten Ressourcen (siehe Kapitel 4.2).

void i8086close()

Gibt alle reservierten Ressourcen frei (siehe Kapitel 4.2).

void i8086reset()

Bringt den Prozessor und die geladenen Geräte (siehe Kapitel 7.5) in einen definierten Standardzustand. Dabei werden alle Register auf 0 gesetzt und die Nachrichtenliste geleert, ohne sie zu verarbeiten.

int i8086loadBinFile(i8086core core, const char* filename, unsigned short beginAdr)*

Lädt eine Datei mit dem Namen *filename* ab Adresse *beginAdr* in den Speicher.

*int i8086execCommand(i8086core *core, i8086command **cmds);*

Führt den Befehl, auf den IP zeigt, aus und setzt IP neu. Die Befehlsnummer ist der Index für das *i8086command* Array. Alle *SYNC_CLOCKS* Takte wird die benötigte Zeit gemessen und daraus berechnet, wie lange der Prozessor pausiert werden muss, um ihn an die Geschwindigkeit des reellen 8086 anzupassen. Nach der Verarbeitung des Prozessorbefehls wird die Nachrichtenliste verarbeitet.

5 Emulatorfunktionen und Opcodes

5.1 Ausgewählte Funktionen der Datei i8086emufuncs.c

5.1.1 Speicherzugriffsfunktionen

typedef unsigned short i808616BitAdr;

*void memWrite(i8086core *core, i808616BitAdr adr, unsigned short val,*
unsigned char w, int reg)
*unsigned short memRead(i8086core *core, i808616BitAdr adr, unsigned char w,*
int reg)

Mit Hilfe dieser zwei Funktionen ist es möglich in den Speicher des i8086 zu schreiben bzw. daraus zu lesen. Bei den Funktionen wird hierzu der *core* (globale Variable), die Adresse (Variable vom Typ *i808616BitAdr*) an der gelesen bzw. geschrieben werden soll, das Segmentregister (Konstanten aus *i8086proc.h*) welches benutzt werden soll, sowie die Variable *w* (*w*=0 - 8 Bit lesen/schreiben, *w*=1 - 16 Bit lesen/schreiben) übergeben. Der Funktion *memWrite* wird zusätzlich noch die Variable *val* übergeben, in der der zu schreibende Wert stehen sollte. Der Rückgabewert von *memRead* stellt das byte bzw. word an der gelesenen Speicheradresse dar.

Beispiel:

Um einen 16 Bit Wert von der Adresse 0x0100 unter der Verwendung des Segmentregisters DS zu lesen und nach 0x0200 zu schreiben würden folgende Aufrufe gemacht werden:

```
i808616BitAdr adr=0x0100;
unsigned short value;

value=memRead(core,adr,1,i8086_REG_DS);
adr=0x0200;
memWrite(core,adr,value,1,i8086_REG_DS);
```

5.1.2 Speicheradresse eines Befehls dekodieren

*i808616BitAdr decodeMemAdr(i8086core *core, char modrm, signed short disp)*

Mit dieser Funktion kann die Speicheradresse berechnet werden, die in einem Maschinenbefehl angesprochen werden soll. Zu übergeben ist der core, das 1. Parameterbyte des Befehls (*modrm*) und das Displacement (2. und 3. Parameter-Byte des Befehls).

Beispiel:

MOV [0x0100],bx

im Maschinencode wäre das: 891E0001

Der Aufruf um die Adresse zu erhalten wäre:

```
i808616BitAdr adr;
adr=decodeMemAdr(core, para.b[0],joinBytes(para.b[1],para.b[2]));
```

5.1.3 Funktionen zum Zugriff auf die I/O-Ports

```
unsigned char portOpByte(i8086core *core,unsigned char type, unsigned char
                         data, unsigned short portNum, unsigned char rw)
unsigned short portOpWord(i8086core *core, unsigned char type, unsigned short data,
                          unsigned short portNum, unsigned char rw)
```

Diese beiden Funktionen werden benutzt um ein Byte bzw. ein Word von den Ports zu lesen bzw. in die Ports zu schreiben. Für den Parameter type gibt es die beiden möglichen Konstanten *i8086_INPUT_PORT* und *i8086_OUTPUT_PORT*, da die I/O Ports physisch getrennt sind. Der Parameter rw gibt an ob vom Port gelesen bzw. in den Port geschrieben werden soll. Hierzu gibt es ebenfalls zwei Konstanten *i8086_READ_PORT* (zum Lesen) und *i8086_WRITE_PORT* (zum Schreiben). Der Parameter *data* wird nur ausgewertet, wenn eine Schreiboperation durchgeführt wird und enthält den zu schreibenden Wert.

Beispiel:

Kopieren eins Bytes von INPUT-Port 10 nach OUTPUT-Port 99

```
unsigned char byte;
byte= portOpByte(core, i8086_INPUT_PORT, 0, 10, i8086_READ_PORT);
portOpByte(core,i8086_OUTPUT_PORT,byte,99,i8086_WRITE_PORT);
```

5.1.4 Stackoperationen

```
void push(i8086core *core, unsigned short x)
unsigned short pop(i8086core *core)
```

Um den Stack des i8086 zu benutzen, stehen die zwei Funktionen *push* und *pop* zur Verfügung. Beiden muß der *core* übergeben werden - bei *push* wird der Wert, der auf den Stack gelegt werden soll im Parameter *x* übergeben.

5.1.5 Funktionen zur Korrektur des Flagregisters

*void correctOvCaFlagafterAddSub(i8086core *core, unsigned short a, unsigned short b,*
unsigned short c, unsigned char w,
unsigned char addsub)
*void correctArithmeticFlags(i8086core *core, unsigned short i,unsigned char w)*

Um nach einer Addition oder Subtraktion das Overflow- und Carryflag zu korrigieren, übergibt man der Funktion *correctOvCaFlagafterAddSub* den *core*, den ersten Summanden (*a*), den zweiten Summanden (*b*), das Ergebnis, die Operationsgröße (*w*=0-8Bit, *w*=1-16Bit) und *addsub* (*addsub*=1 - Addtition, *addsub*=0 - Subtraktion).

Um das Zero-, Parity- und Signflag anhand eines Wertes zu setzen, verwendet man die Funktion *correctArithmeticFlags* und übergibt ihr den *core*, den Wert nachdem die Register gesetzt werden sollen (*i*) und die Operationsgröße (*w*=0-8 Bit, *w*=1-16Bit).

5.1.6 Einfügen eines neuen Opcodes

Angenommen der Emulator soll um einen Opcode und damit um einen neuen Maschinenbefehl erweitert werden. Dazu muss zunächst eine Emulatorfunktion für diesen Maschinenbefehl geschrieben werden. Diese muss den folgenden Prototypen aufweisen:

*void (*i8086NewOpcode)(i8086core *core, unsigned char opcode,i8086Parameter para,*
i8086Parameter data)

Ist diese Funktion implementiert, so muss diese noch in der Datei i8086emufuncs.c in der Funktion *i8086InitCommands(i8086command **cmds)* mit folgendem Befehl registriert werden.

```
i8086command* i8086AddCommnand(char name[i8086_OPC_NAME_LEN],
                               unsigned char opcode,
                               unsigned char size, unsigned char hasMod,
                               unsigned char clocks,
                               void (*func)(i8086core *core,
                                            unsigned char command,
                                            i8086Parameter para,
                                            i8086Parameter data),
                               i8086command **cmds
                               )
```

Beispiel:

```
/*                    Name        OpC Sz Mod  Clk   EmuFkt              cmds */
i8086AddCommand("NewOpc" , 242, 2, 0,      8,    i8086NewOpcode,  cmds);
```

Hierbei kann *Name* ein beliebiger String mit der max. Länge *i8086_OPC_NAME_LEN* sein, *OpC* sollte der neue Opcode (dezimal) sein und *Size* die minimale Größe des Maschinenbefehls inkl. Parameter in Bytes. Sollte der neue Befehl unterschiedliche Größen haben z.B. weil er auf eine Speicheradresse zugreift, so muss *Mod* auf 1 gesetzt werden (sonst auf 0). *Clk* gibt die Anzahl der Takte an, die die Ausführung des Befehls in Anspruch nehmen soll. Das war's, der neue Befehl ist nun im Prozessor implementiert.

5.2 Ausgewählte Makros der Datei i8086gui_error.h

*void i8086guiMessage(GtkWindow *parent, const char *msg, int mode)*
*void i8086guiError(GtkWidget *parent, const char *msg1, const char *msg2)*

Zur Fehlerbehandlung bzw. Nachrichtenanzeige stehen 2 Makros zur Verfügung. *i8086guiMessage* zeigt dem User ein GTK-Dialogfenster mit der übergebenen *msg* als Inhalt. Der Parameter *mode* legt das Icon fest, welches angezeigt werden soll. Für *mode* stellt GTK folgende Konstanten zur Verfügung:
GTK_MESSAGE_INFO, *GTK_MESSAGE_WARNING*, *GTK_MESSAGE_QUESTION*, *GTK_MESSAGE_ERROR*.

Das Makro *i8086guiError* erstellt ebenfalls ein Dialogfenster und gibt die übergebenen Parameter *msg1* und *msg2* getrennt durch einen Zeilenumbruch aus. Weiterhin schreibt die Funktion die Meldung als "Error" in das Logfile, welches durch *i8086_LOG_FILE* in der Datei *i8086error.h* definiert ist. Nachdem das Dialogfenster der Fehlermeldung geschlossen wurde beendet *i8086guiError* den Emulator.

5.3 Ausgewählte Makros der Datei i8086error.h

void i8086error(msg1, msg2)
void i8086warning(msg)
void i8086clearLog()

Diese beiden Makros entsprechen denen aus *i8086gui_error.h* (siehe Kapitel 5.3) , werden jedoch nur für die Konsolenversion des Emulators (*i8086text.c*) verwendet. *i8086clearLog()* leert das Logfile, welches durch *i8086_LOG_FILE* in der Datei *i8086error.h* definiert ist. *i8086warning* schreibt msg mit der Zeilennummer und der Datei, in der es aufgerufen wurde, in das Logfile. In der Datei *i8086error.h* sind außerdem die Standardfehlermeldungen definiert.

5.4 Ausgewählte Funktionen der Datei i8086config.c

*int i8086ReadStrConfig(char *buf, const char *filename, const char *name)*
*int i8086ReadHexConfig(const char *filename, const char *name, int defValue)*
*int i8086ReadDecConfig(const char *filename, const char *name, int defValue)*
*int i8086ReadBoolConfig(const char *filename, const char *name, int defValue)*
*int i8086NameExistsConfig(const char *filename, const char *name)*

Zum Arbeiten mit der Konfigurationsdatei, welche in i8086config.h als CONFIG_FILE definiert wurde, gibt es einige Funktionen. Bei allen Funktionen bezeichnet *filename* den Dateinamen von dem gelesen werden soll. Zum Auslesen eines Strings aus der Konfigurationsdatei wird *i8086ReadStrConfig* benutzt. Sie liest den zu *name* passenden Eintrag als einen String in den übergebenen Parameter *buf*. Die Funktionen *i8086ReadHexConfig*, *i8086ReadDecConfig* und *i8086ReadBoolConfig* lesen einen Hex-, Dezimal bzw. Booleanwert aus der Datei und geben diesen bei Erfolg zurück. Tritt ein Fehler beim Lesen auf, wird der in *defValue* übergebene Wert zurückgegeben. Zum Prüfen ob ein Eintrag in der Datei existiert kann die Funtkion *i8086NameExistsConfig* genutzt werden. Ist der im Parameter *name* übergebene Wert in der Datei vorhanden, wird 1 zurückgegeben - sonst 0.

6 Messages

6.1 Übersicht

Die Kommunikation zwischen Geräten untereinander und Geräten mit dem Prozessor wird über Nachrichten realisiert. Die Speicherung der Nachrichten in einer Liste und deren Verwaltung erfolgt durch den Prozessor. Nach jedem Prozessortakt wird die gesamte Liste verarbeitet und danach geleert. Eine Nachricht besteht aus einer Nachrichtennummer und zwei Nachrichtendatensätzen.

Alle Funktionen, die mit der Nachrichtenliste arbeiten, sind durch Semaphoren geschützt und können somit auch für parallele Threads verwendet werden. Es können maximal *i8086_MSG_QUEUE_LEN* in der Nachrichtenliste gespeichert werden. Ist die Liste voll, werden einzufügende Nachrichten ignoriert. Die Anzahl verschiedener Nachrichtennummern ist begrenzt auf *i8086_MAX_MSG*. Zu jeder Nachrichtennummer kann eine Behandlungsroutine gesetzt werden. Wird eine Nachricht vom Prozessor verarbeitet, ruft dieser die entsprechende Routine auf. Da für jede Nummer nur eine Funktion zugewiesen werden kann, müssen die Behandlungsroutinen die vorherigen Funktionen für diese Nachrichtennummer selbst aufrufen. In einer Nachrichtenliste darf immer nur eine Interrupt-Nachricht (*i8086_SIG_CALL_INT*) enthalten sein.

#include “i8086messages.h“

6.2 Nachrichten-Funktionen

```
typedef struct i8086Msg
{
  unsigned int msg;        /* Nachrichtennummer  */
  unsigned int hParam;   /* Daten der Nachricht  */
  unsigned int lParam;   /* Daten der Nachricht  */
}i8086Msg;

typedef void (i8086msgFunc)(unsigned short msg, unsigned int hParam,
                             unsigned int lParam)
typedef i8086msgFunc* (i8086PSetMsgFunc)(unsigned short msg,
                                         i8086msgFunc *msgFunc)
```

void i8086initMsgQueue()

Initialisiert die Nachrichtenliste und alle Ressourcen, die für die Arbeit mit ihr benötigt werden. Sollte nur für interne Aufgaben verwendet werden.

void i8086PushMsg(unsigned short msg, unsigned int hParam, unsigned int lParam)

Fügt eine neue Nachricht mit der Nummer *msg* an das Ende der Liste ein. Mit *hParam* und *lParam* können nachrichtenspezifische Daten übertragen werden.

i8086Msg i8086PopMsg()*

Gibt die erste Nachricht der Liste zurück und entfernt die Verwaltungsdaten. Die zurückgegebene Nachricht muss nach ihrer Verwendung freigegeben werden. Sind keine weiteren Nachrichten in der Liste wird NULL zurückgegeben.

i8086msgFunc i8086SetMsgFunc(unsigned short msg, i8086msgFunc *msgFunc)*

Setzt die Behandlungsroutine für die Nachricht *msg*. Der Rückgabewert ist die Routine, die zuvor mit dieser Nummer verknüpft war. War keine Routine vorhanden wird *NULL* zurückgegeben.

void i8086ClearMsgQueue()

Löscht die gesamte Nachrichtenliste ohne sie zu verarbeiten. Sollte nur für interne Aufgaben verwendet werden.

*void i8086ProcessMsgQueue(i8086core *core)*

Verarbeitet die Nachrichten in der Nachrichtenliste und löscht sie danach. Sollte nur für interne Aufgaben verwendet werden.

void i8086CloseMsgQueue()

Löscht die gesamte Nachrichtenliste ohne sie zu verarbeiten und gibt danach alle für die Arbeit mit der Liste benötigten Ressourcen frei. Sollte nur für interne Aufgaben verwendet werden.

6.3 Versenden einer Nachricht

Für das Senden einer Nachricht steht die Funktion *i8086PushMsg* zur Verfügung. Folgende Nachrichten können verwendet werden.

#define i8086_SIG_CALL_INT 0

Löst den Interrupt mit der in *hParam* übergebenen Nummer aus. Es sollte immer nur ein Interrupt pro Prozessortakt eingefügt werden (siehe *i8086_SIG_INT_REQUEST*).

#define i8086_SIG_PORT_OUT 1

Die Behandlungsroutine für diese Nachricht wird immer aufgerufen wenn der Wert eines Ports geändert wurde. In *hParam* steht die Nummer des Ports dessen Wert gesetzt wurde.

#define i8086_SIG_PORT_WRITE_VALUE 2

Schreibt in den Port *hParam* den Wert *lParam*.

#define i8086_SIG_INT_REQUEST 3

Die Behandlungsroutine für diese Nachricht wird nach jedem Prozessortakt aufgerufen. In *hParam* stehen die bereits in die Nachrichtenliste eingefügten Interrupts. Der erste Aufruf der Routine erfolgt durch den Prozessor. Bei Kaskadierten Aufrufen muss *hParam* in der jeweiligen Funktion erhöht werden, wenn diese einen Interrupt eingefügt hat. In *lParam* befinden sich die aktuellen Flags des Prozessors.

#define i8086_SIG_IRQ 4

Wird ausgelöst wenn ein Gerät einen IRQ gesendet hat. Die IRQ-Maske wird in *hParam* übergeben. Dazu können folgende IRQs mit einander kombiniert werden.

```
#define i8086_IRQ_0 1
#define i8086_IRQ_1 2
#define i8086_IRQ_2 4
#define i8086_IRQ_3 8
#define i8086_IRQ_4 16
#define i8086_IRQ_5 32
#define i8086_IRQ_6 64
#define i8086_IRQ_7 128
```

#define i8086_SIG_WMEM 6

Schreibt das Byte in *lParam* an die Adresse in *hParam*.

#define i8086_SIG_WBMEM 7

Schreibt einen Block, mit einer max. Länge von 4095 Bytes in den Speicher des Emulators. In den Bits 0-19 von *hParam*, befindet sich die Adresse ab der geschrieben wird und in den Bits 20-31 die Anzahl der zu übertragenden Bytes. *lParam* beschreibt die Adresse, des Speichers aus dem gelesen wird. Mit dem Makro *GetAdrAndCount(int, int)* kann der Wert für *hParam* erstellt werden.

Beispiel:

Das folgende Beispielschreibt 4000 Zufallszahlen, ab Adresse 0x1000 in den Speicher.

```
#define m_size 4000
char *mem;
int i;
```

```
mem = (char*)malloc(m_size);
srand(time(NULL));
for (i=0; i<m_size; i++)
  mem[i] = (char unsigned)rand();
funcCalls->SendMsg(i8086_SIG_WBMEM, GetAdrAndCount(0x1000, m_size),
                        (unsigned int)mem);
```

#define i8086_SIG_WBMEM_EXEC 8

Wird vom i8086_SIG_WBMEM-Handler gesendet, nachdem der Speicherinhalt kopiert wurde. *hParam* und *lParam* haben die Werte der zugehörigen i8086_SIG_WBMEM-Nachricht.

#define i8086_SIG_LOAD_FILE 9

Lädt eine Datei in den Speicher. In *hParam* steht die Adresse(*char**) des Dateinamens, in *lParam* die Adresse ab der die Datei in den Speicher geschrieben wird.

Beispiel:

```
static char filename[] = "file.bin";
funcCalls->SendMsg(i8086_SIG_LOAD_FILE, (unsigned int)filename, 0x1000);
```

#define i8086_SIG_USER 15

Diese Nachricht und folgende sind für die freie Verwendung gedacht.

6.4 Empfangen einer Nachricht

Um über das Eintreffen einer bestimmten Nachricht informiert zu werden, steht die Funktion *i8086SetMsgFunc* zur Verfügung. Nach jedem Prozessortakt werden die Nachrichten verarbeitet und die mit *i8086SetMsgFunc* registrierten Funktionen der jeweiligen Nachricht aufgerufen. Eine neue Behandlungsroutine wird eingefügt indem *i8086SetMsgFunc* mit der Nachrichtennummer und der Adresse der Behandlungsroutine aufgerufen wird. Der Rückgabewert von *i8086SetMsgFunc* ist die Funktion die zuvor mit dieser Nachrichtennummer verknüpft war. Damit alle Behandlungsroutinen aufgerufen werden, müssen die zurückgegebenen Funktionen von den aktuell registrierten aufgerufen werden.

Beispiel:

In diesem Beispiel wird eine Behandlungsroutine für die Änderung eines Ausgangsports registriert. Die alte Routine wird gespeichert und in der neuen aufgerufen.

```
#include "i8086messages.h"

i8086msgFunc *OldPortOutHandler=NULL;
...

void PortOutHandler(unsigned short msg, unsigned int hParam, unsigned int lParam)
{
    ...
    if (OldPortOutHandler != NULL)
        OldPortOutHandler(msg, hParam, lParam);
}

void sample_init()
{
    ...
    OldPortOutHandler = i8086SetMsgFunc(i8086_SIG_PORT_OUT, PortOutHandler);
    ...
}

...
```

7 Devices

7.1 Übersicht

Der i8086-Emulator bietet die Möglichkeit an, zusätzliche Geräte einzubinden. Die Gerätedateien werden als dynamische Bibliotheken erstellt und kommunizieren mit dem Prozessor über Nachrichten (siehe Kapitel 6), welche nach jedem Prozessortakt verarbeitet werden.

#include "i8086devices.h"
Benötigte Objektdateien: i8086messages.o

7.2 Geräte-Interface

Das Geräte-Interface besteht aus drei Funktionen.

```
typedef void (i8086msgFunc)(unsigned short msg, unsigned int hParam,
                            unsigned int lParam);
typedef i8086msgFunc* (i8086PSetMsgFunc)(unsigned short msg,
                          i8086msgFunc *msgFunc);
typedef unsigned char (i8086MemReadByte)(i808616BitAdr adr, i808616BitAdr seg);
typedef unsigned short (i8086MemReadWord)(i808616BitAdr adr, i808616BitAdr seg);

typedef struct i8086DeviceCalls
{
  char* Version;
  i8086PSetMsgFunc *SetMsgHandler;
  i8086msgFunc *SendMsg;
  i8086MemReadByte *MemReadByte;
  i8086MemReadWord *MemReadWord;
}i8086DeviceCalls;
```

*void DeviceInit(i8086DeviceCalls *calls)*

Die DeviceInit-Funktion wird beim Laden des Gerätes aufgerufen. Der Parameter dieser Funktion enthält Adressen zu Funktionen für die Kommunikation mit dem Prozessor. Der struct i8086DeviceCalls ist in der Datei i8086devices.h definiert. Die genaue Beschreibung der beiden Nachrichtenfunktionen ist im Kapitel 6.2 zu finden. Der Zeiger auf i8086DeviceCalls wird nicht automatisch freigegeben und sollte deshalb bei Bedarf vom entsprechenden Geräte aus dem Speicher entfernt werden. Diese Funktion muss in jedem Gerät definiert sein.

void DeviceClose()

DeviceClose wird aufgerufen, wenn der Prozessor das Gerät nicht mehr benötigt und es aus der Geräteliste entfernt wird. In dieser Funktion sollten alle vom Gerät verwendeten Ressourcen freigegeben werden. Diese Funktion muss nicht definiert werden.

void DeviceReset()

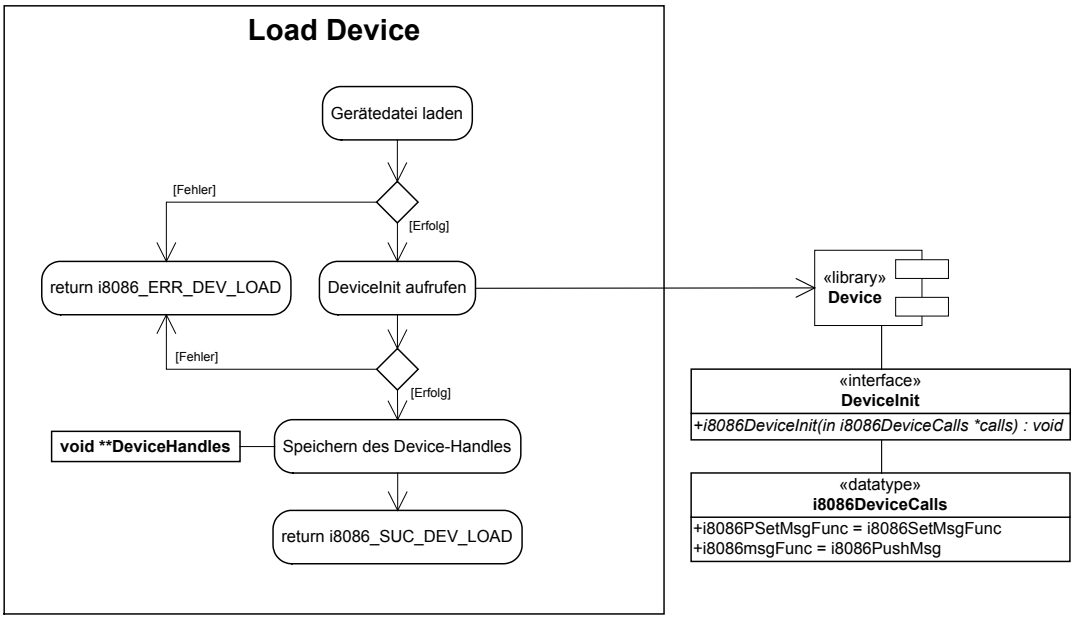
DeviceReset wird bei jedem Reset des Prozessors aufgerufen. Mit dieser Funktion sollte das Gerät in einen Standardzustand gebracht werden. Diese Funktion muss nicht definiert werden.

7.3 Kommunikation

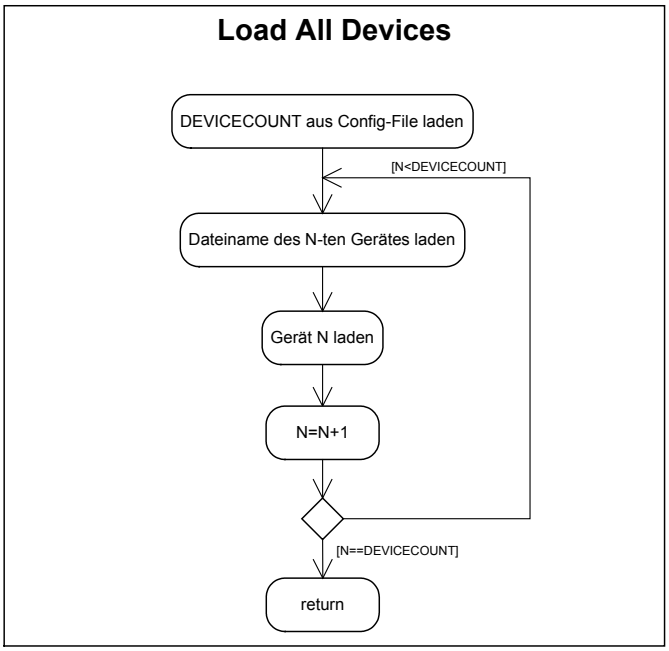
Die Kommunikation mit anderen Geräten und mit dem Prozessor erfolgt über eine MessageQueue. Die Funktionen für die Arbeit mit der MessageQueue sind semaphorisch gesichert und daher auch für Geräte mit eigenem Thread verwendbar. Die genaue Beschreibung der Nachrichtenfunktionen ist im Kapitel 6.2 zu finden.

7.4 Laden der Geräte

Alle Geräte die in der Konfigurationsdatei (siehe Kapitel 2.4.2) eingetragen sind, werden beim Start des Prozessors (siehe Kapitel 4.2) geladen. Beim Laden wird die in dem Gerät deklarierte Funktion *DeviceInit* (siehe Kapitel 7.2) aufgerufen. Die Geräte werden nacheinander und in der Reihenfolge in der sie in der Konfigurationsdatei stehen geladen. Die folgenden Diagramme zeigen die interne Arbeitsweise des Prozessors beim Laden eines einzelnen Gerätes und aller Geräte.



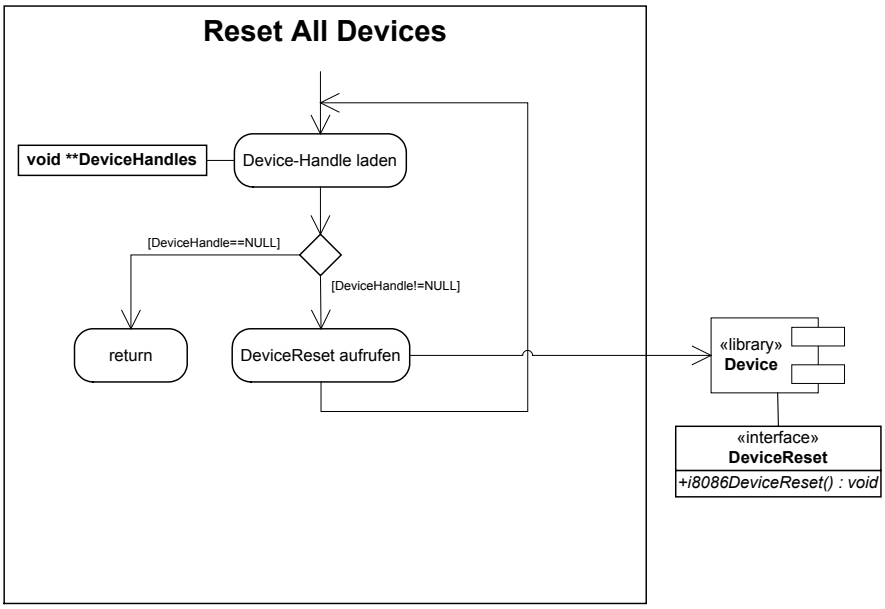
Laden eines einzelnen Gerätes



Laden aller Geräte

7.5 Zurücksetzen der Geräte

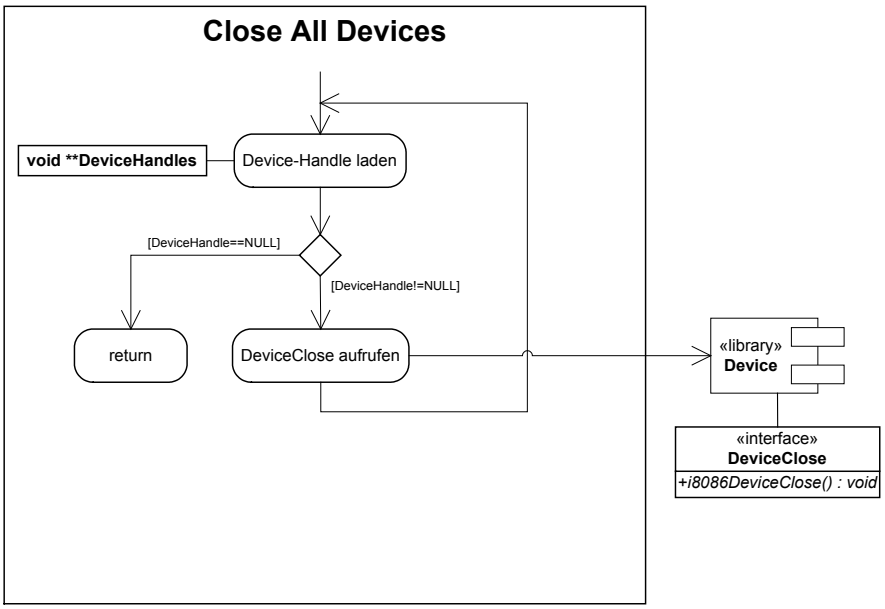
Bei einem Reset des Prozessors wird in allen Geräten die Funktion *DeviceReset* (siehe Kapitel 7.2) aufgerufen. Kann die *DeviceReset*-Funktion nicht gefunden werden wird das Gerät ignoriert und mit dem nächsten fortgefahren. In dieser Funktion sollte das Gerät in einen Standard-Zustand gebracht werden.



Zurücksetzen aller Geräte

7.6 Schliessen der Geräte

Beim Schließen des Prozessors werden ebenfalls alle Geräte geschlossen. Hierfür wird die Funktion *DeviceClose* (siehe Kapitel 7.2) aufgerufen. In dieser Funktion sollten alle vom Gerät verwendeten Ressourcen freigegeben werden.



Schließen aller Geräte

7.7 Arbeiten mit den vorhandenen Geräten

Der i8086emu verfügt bis auf den Parallel- und RS-323-Port über die gleichen Peripheriebausteine wie der SBC86 Singleboardcomputer. Darunter finden sich 8 LEDs, 8 Schalter, 8 7-Segment-Anzeigen sowie eine Tastatur. Für die Interruptsteuerung gibt es außerdem einen eingeschränkten PIC (Programmable Interrupt Controller) und einen PIT (Programmable Interval Timer). Auf die auf dem echten SBC86 verfügbaren seriellen und parallelen Ports wurde verzichtet.

7.7.1 LEDs

Die LEDs lassen sich über den Port 0h der CPU digital ansprechen.

Beispiel

```
mov al,0aah
out 0,al
```

Die LEDs sollten 1010 1010 anzeigen (1 LED an, 0 LED aus).

7.7.2 Schalter

Die Schalter werden analog zu den LEDs digital über den Port 0h eingelesen.

Beispiel:

Schalterstellung: 1010 1010 (1 an, 0 aus).

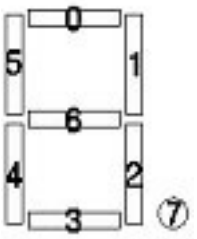
```
in al,0
```

AL sollte nun 0aah enthalten.

7.7.3 7-Segment-Anzeigen

Die 7-Segment Anzeigen sind mit den Ports 90h-9eh (nur die geraden Portnummern) verbunden. Sie lassen sich direkt durch Portoperationen ansprechen, einfacher ist es jedoch dazu die vom ROM des SBC86 bereitgestellten Interruptserviceroutinen zu verwenden, dazu später mehr.

Bei direktem Zugriff entspricht jedes Bit des ausgegebenen Wertes einem Segment (1 Segment leuchtet, 0 aus).



Port 90h entspricht der rechten Anzeige, 92h der 2. Anzeige von rechts usw.

7.7.4 Tastatur

Die Tastaturmatrix kann über den Port 80h abgefragt werden. Bei jedem Lesezugriff ändert sich der gelesene Wert solange keine Taste gedrückt wird. Bleibt bei mehreren Zugriffen der Wert gleich wird eine Taste betätigt.

Der Tastencode ist wie folgt aufgebaut

0	0	Spalte	Spalte	Spalte	Zeile	Zeile	Zeile	
					1	1	0	Zeile 0 (obere)
					1	0	1	Zeile 1
					0	1	1	Zeile 2
		0	0	0				Spalte 0 (linke)
		0	0	1				Spalte 1
		0	1	0				Spalte 2
								...
		1	1	1				Spalte 7

7.7.5 Display/Keyboard Zugriff über Interrupts

Wie bereits erwähnt stellt das ROM des SBC-86 ein paar nützliche Routinen zur einfachen Ein- und Ausgabe bereit.

INT 5 - Tastaturstatus ermitteln

Parameter: AH = 0
Rückgabe: AL = 00 keine Taste betätigt,sonst Taste betätigt

INT 5 - Warten auf Tastenbetätigung

Parameter: AH = 1
Rückgabe: AL = Tastencode
0..F 00..0F
ENTER 10
G 11
S 12 usw.

INT 5 - Eingabe einer 16-Bit-Hexadezimalzahl

Parameter: AH = 2
BX = Vorgabewert
DL = Displaystelle
Rückgabe: AX = eingegebene Hexadezimalzahl
DL = Tastaturcode der Quittung (10h, 16h oder 17h)
Der Vorgabewert wird ab der gewünschten Stelle nach rechts ins Display geschrieben und kann dann über die Tastatur geändert werden.

INT 6 - Display löschen

Parameter: AH = 0

Das gesamte Display wird gelöscht.

INT 6 - Ausgabe eines ASCII-Zeichens

Parameter: AH = 1
AL = Zeichen
DL = Displaystelle

Das ASCII-Zeichen wird in die gewünschte Displaystelle geschrieben. Die Displaystellen werden von rechts mit 0 beginnend numeriert.

INT 6 - Ausgabe eines ASCII-Zeichenkette

Parameter: AH = 2
BX = Adresse des 1.Zeichens
DL = Displaystelle

Die Zeichenkette wird ab der gewünschten Stelle nach rechts ins Display geschrieben. Als Textendekennung wird 00 erwartet.

INT 6 - Ausgabe einer 16-Bit-Hexadezimalzahl

Parameter: AH = 3
BX = Hexadezimalzahl
DL = Displaystelle

Die angegebene Zahl (4 Hexzeichen) wird ab der gewünschten Stelle nach rechts ins Display geschrieben.

INT 6 - Ausgabe einer 8-Bit-Hexadezimalzahl

Parameter: AH = 4
BL = Hexadezimalzahl
DL = Displaystelle

Die angegebene Zahl (2 Hexzeichen) wird ab der gewünschten Stelle nach rechts ins Display geschrieben.

7.7.6 Programmable Interrupt Controller 8259A

Auf eine genaue technische Beschreibung des PIC wird an dieser Stelle verzichtet, entsprechende Informationen sind frei verfügbar. Prinzipiell dient der PIC dazu, die Interruptanforderungen von Peripherie-Geräten zu verarbeiten und an die CPU weiterzuleiten.

Der emulierte PIC besitzt nicht die volle Funktionalität des 8259A. Angeschlossen ist er an den Ports C0h und C2h. Er verfügt über keine Prioriätensteuerung und unterstützt nur den 8086 Mode ohne Kaskadierung. Der Auto- und Normal-EOI Mode, sowie das IRQ-Maskenregister ist vorhanden.

Folgendes Beispiel aktiviert den IRQ0 des Maskenregisters so das dieser vom 1. Timer-Kanal benutzt werden kann

```
in al,0c2h          ;Lesen des Int.-Maskenregisters des PIC
and al,11111110b
out 0c2h,al         ;Setzen des Int.-Maskenregisters des PIC
```

Da der PIC normalerweise Im AOI-Modus initialisiert wird muss nach jedem ausgeführten IRQ der PIC wieder freigegeben werden:

```
mov al,20h
out 0c0h,al
```

Programmierbeispiele liegen dem Emulator bei.

7.7.7 Programmable Interval Timer 8253

Der PIT ermöglicht die Programmierung zeitgesteuerter Abläufe.
Die entsprechenden Ports sind A6h und A2h.

Einschränkungen

Von den eigentlich 3 verfügbaren Timer-Kanälen sind nur der erste (IRQ0) und der zweite (IRQ1) an den PIC angeschlossen. Eine grundlegende Initialisierung wird emuliert, sowie ausschließlich der Zählermodus 3. Die Taktfrequenz sollte 1,8432 MHz betragen, allerdings lässt sich eine solche Auflösung systembedingt im Emulator nicht erreichen. Der emulierte PIT zählt pro Schritt 18432 mal, d.h. das kleinstmögliche erreichbare Zeitintervall ist 1/(1,8432 Mhz/18432)=1/100 Sekunde. Die durchschnittliche Geschwindigkeit wird jedoch relativ genau emuliert.

Dieses Beispiel progammiert den 1.Kanal des PIC auf eine Frequenz von 100 Hz, d.h. das der PIC alle 10ms IRQ0 auslöst.

```
zk equ 18432          ;Zeitkonstante fuer 10-ms-Interrupt
mov al,01110110b
out 0a6h,al           ;Zaehler 1 im Mode 3
mov al,zk&00FFh
out 0a2h,al           ;Low-Teil der Zeitkonstante laden
mov al,zk>>8
out 0a2h,al           ;Hi-Teil der Zeitkonstante laden
```

Programmierbeispiele (z.b. Clockmem) liegen dem Emulator bei.

7.7.8 Anhang

Interruptvektortabelle, IRQs

0000H	INT 0	Divisionsüberlauf	nicht genutzt
0004H	INT 1	Single Step Interrupt	Monitor
0008H	INT 2	NMI	nicht genutzt
000CH	INT 3	Breakpoint Interrupt	Monitor
0010H	INT 4	INTO	nicht genutzt
0014H	INT 5	Tastaturzugriff	Monitor
0018H	INT 6	Displayzugriff	Monitor
001CH	INT 7		frei
0020H	INT 8	IRQ0 PIT Kanal 1	nicht genutzt
0024H	INT 9	IRQ1 PIT Kanal 2	nicht genutzt

8 Monitorprogramm (ROM)

8.1 Speicherbereiche

0000h - 003Fh	Interruptvektortabelle
0040h - 004Bh	System Zellen
004Ch - 00FFh	Stack
0100h - 2FFFh	Progamm
C000h - CFFFh	ROM

8.2 Debugger

Das Monitorprogramm (ROM) initialisiert die IVT (s.o.) und stellt einen Debugger zur Verfügung (dessen Benutzung sich im Emulator zwar erübrigt aber dennoch möglich ist). Hier soll nur eine kurze Übersicht gegeben werden. Nach dem Laden eines Programms drückt man “CPU-Reset” und findet sich daraufhin im Debugger Modus des Monitorprogramms wieder.

Über die untere Tastenreihe lässt sich folgende Funktionalität nutzen:

Taste	Funktion
A-F,0-9	Eingabe von Hexadezimalzahlen
E	Enter (Eingabebestätigung) Auch zum Verlassen der aktuellen Funktion.
-	Speicher ansehen und ändern. Es wird eine Startadresse abgefragt, dann über + und – durch den Speicher schalten
+	Register ansehen und ändern. Das Register wird über A-F und 0-9 gewählt + und – schalten zwischen Registern um.
O	Ausgabe eines Wertes auf einem Port.
I	Einlesen eines Wertes von einem Port.
T	Trace Into, ein vorher geladenes Programm Schritt für Schritt abarbeiten.
S	Step Over, im Unterschied zu Trace werden hier Unterprogramme und bedingte Programmschleifen insgesamt abgearbeitet.
G	Go, ein vorher geladenes Programm neu starten, mit der Möglichkeit einen Breakpoint zu setzen.

9 SBC86 Beispiele

Um die Arbeit mit dem SBC86 Emulator zu vereinfachen, haben wir ein paar einfache Beispiele zum besseren Verständnis beigelegt.

9.1 Lauflicht (kitt)

Als einfachstes Beispiel haben wir ein Lauflicht erstellt. An diesem Beispiel kann man die Zugriffe auf die LED-Anzeige und die darunter liegenden Schalter sehen.

Starten Sie den Emulator und laden Sie die Datei Kitt. Jetzt muss nur in den RUN-Modus gewechselt werden und in der LED-Anzeige ist ein Lauflicht zu sehen. Mit dem Schalter 0 aus den Switches kann das Lauflicht verändern werden. Sobald der Schalter 0 gesetzt ist läuft das Lauflicht abwechselnd von links nach rechts und rechts nach links. Wenn der Schalter 0 wieder zurückgesetzt wird, bewegt sich das Lauflicht nur noch in die zuletzt gelaufene Richtung.

Beispiel für die Erstellung

> nasm ./kitt.asm

9.2 Uhr mit Speicher Anzeige (clockmem)

In diesem Beispiel wurde eine Interrupt Service Routine geschrieben und der Timer des SBC86 genutzt. Ebenfalls kann man die acht 7- Segment Anzeige und das Keyboard in Aktion sehen.

Starten Sie den Emulator und laden Sie die Datei clockmem. Jetzt müssen Sie nur noch in den RUN- Modus wechseln und Sie sehen eine Uhr auf den acht 7- Segment Anzeigen. Jetzt kann über die Tasten des Keyboards - O- für Stunde, I- für Minute und T- für Sekunden die aktuelle Uhrzeit eingestellt werden. Mit den Tasten + und – kann der Wert verstellt werden, der aktuell angezeigt wird. Durch die Taste R wird der Vorgang Uhr stellen beendet. Nun kommt der Bereich Speicheranzeige, hier wird über die Zahlen des Keyboards die Speicheradresse eingegeben, die angezeigt werden soll. Mit der Taste R wird auch dieser Vorgang beendet.

Jetzt befinden Sie sich im Hauptprogramm und sehen in diesem Moment den Inhalt der so eben eingegebenen Speicheradresse. Über die Tasten + und – können die Speicheradressen hoch und runter geblättert werden. Mit der Taste G wird in den Uhr Modus gewechselt. Durch Drücken einer beliebigen Taste, kommen Sie zurück in den Speicher-Anzeige-Modus. In diesem Modus kann mit der Taste E erneut die Speicheradresse angegeben werden.

Beispiel für die Erstellung

> nasm ./clockmem.asm

10 Quellen

- Skript Maschinennahe Programmierung in C, Prof. Geiler
- Assemplerprogrammierung mit dem Singleboardcomputer SBC 86, Prof. Beierlein
- Microsoft, Microsoft MASM Reference, Redmond 1992
  Document No. DB35749-1292
- The Intel Microprocessors 8086/8088, 80186/80188, 80286, 80386, 80486
  Architecture, Programming, and Interfacing
  second edition
  by Barry B. Brey
  ISBN 0-675-21309-6
- Der 16-Bit-Mikroprozessor des ESER-PC
  Bonitz
  ISBN 3-341-00704-0
- Das Lehrbuch zur Assemblerprogrammierung der 8086/8088/80186/80286 - Prozessoren unter MSDOS
  4. Auflage 1990
  Bertram Wohak
- 231456.pdf
- 8259A_PIC_Datasheet.pdf

- www.developer.gnome.org
- www.gtk.org
- www.xchat.org
- nsis.sourceforge.net/home/ (installer)
- http://web.sfc.keio.ac.jp/~s01397ms/cygwin/index.html.en

10.1 Sourcecode von Dritten

Folgende Dateien bzw. Funktionen sind von Dritten.

Für den Disassambler des Emulators wurden folgende Dateien eingebunden. Diese Dateien stammen vom NASM (http://nasm.sourceforge.net/).

disasm.c
insnsd.c
names.c
regflags.c
regvals.c
insnsa.c
insnsn.c
regdis.c
regs.c
sync.c

Monitorprogramm SBC86 (Rom.bin) von der Hochschule Mittweida.

Die Funktion *stringcat* in der Datei i8086gui_emufuncs.c ist aus dem Vorlesungsskript von Prof. Geiler (Hochschule Mittweida).

Folgende Funktionen entstammen dem Sourcecode für X-Chat 2.0.6 (c) 1998-2003 von Peter Zelezny und wurden nur für die About-box benutzt (http://www.xchat.org).

get_mhz
get_cpu_str
get_cpu_info
waitline