Ein einfacher Einchipvideorechner

Christian Berger

4. Dezember 2006

Inhaltsverzeichnis

Kapitel 1

Motivation

Ich wollte einen möglichst einfachen Rechner entwickeln, der einfach nachzubauen ist, jedoch mächtig genug, ein ‚Gesicht'zu haben. Viele Rechner in unserer Umgebung werden gar nicht mehr als Rechner wahrgenommen. Sie stecken in Geräten wie Videorekordern, elektrischen Nasenwärmern oder Kaffeemaschinen. Obwohl viele von uns ständig von solchen Geräten umgeben sind, ist das Interesse in sie sehr gering.

Dieses Projekt könnte das Interesse in eingebettete Systeme bei vielen Leuten wecken, in dem es zeigt, was alles möglich ist.

Kapitel 2

Hardware

Wie schon erwähnt sollte die Hardware möglichst einfach sein. Jeder, der weiß, welches Ende des Lötkolbens man anfassen darf, sollte in der Lage sein, dieses Projekt nachzubauen.

Die Hardware besteht eigentlich nur aus dem nötigsten. Ein Quarz wird mit 2 Kondensatoren an den Controller angeschlossen. 2 Widerstände sorgen für die Spannungspegel am Videoausgang, 2 weitere arbeiten als Pull-UPs an der Tastatur.

Der Rest wird in Software realisiert.

2.1 Stromlaufplan

2.2 Layout

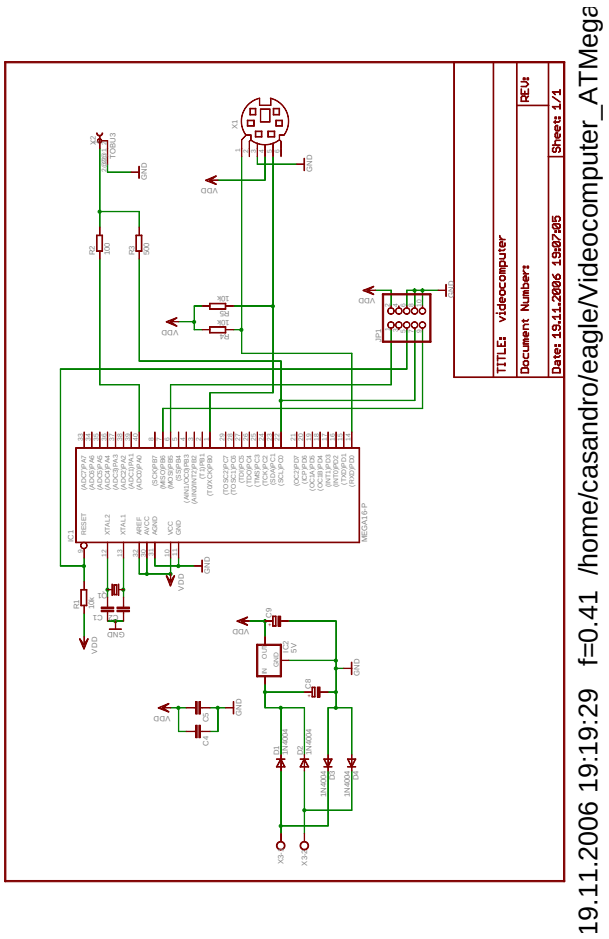

Abbildung 2.1: Stromlaufplan

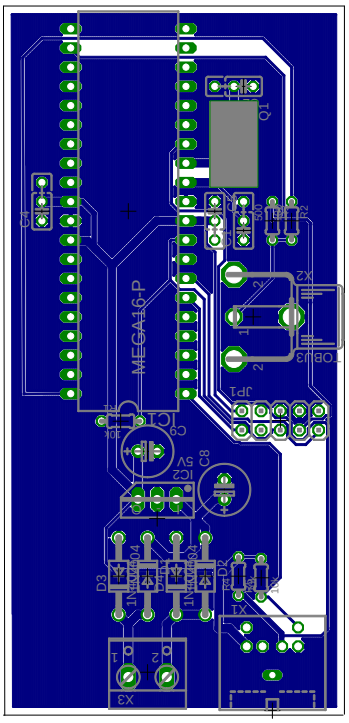

Abbildung 2.2: Layout

Kapitel 3

Software

Die Software ist das eigentlich interessante an diesem Projekt. Ich teile sie, ähnlich dem Userspace-Kernelspace-Konzept von Unix in 2 Bereiche auf.

3.1 Hardwarenahe Schicht

Die hardwarenahe Schicht besteht im Grunde aus Initialisierungsroutinen, die sich darum kümmern, dass die Hardware korrekt initialisiert wird, sowie 2 Unterbrechungsdienstroutinen.

3.1.1 Speicheraufbau

Die verwendeten Controller haben jeweils 1024 Bytes frei benutzbaren Arbeitsspeicher plus 32 Register, sowie mindestens 8 Kilobyte ROM.

Arbeitsspeicher

Der Arbeitsspeicher beinhaltet einige Verwaltungsvariablen, die 20 Bildschirmzeilen, sowie ein Feld mit Zeigern auf diese Bildschirmzeigern. Der genaue Aufbau findet sich in Tabelle 3.1 auf Seite 12.

Dieses Feld ermöglicht es, den Beginn jeder Bildschirmzeile ohne aufwändige Berechnungen zu finden. Zusätzlich kann es auch noch für spezielle Effekte wie beispielsweise das Rollen des Bildschirminhaltes verwendet werden. Dazu müssen nur die 20 Zeiger durchgerollt werden.

Festwertspeicher

Der Festwertspeicher enthält an seinem oberen Ende den Zeichensatz. Dieser ist so organisiert, dass das gewünschte Zeichen über die niederwertigsten 8 Bit ausgewählt werden kann, während die oberen 3-4 Bit den Zeichensatz und die Zeichenzeile auswählen. Dies vermeidet zeitraubende Adressberechnungen-

3.1.2 Tastaturunterbrechungsdienstroutine

Die einfachere Routine kümmert sich um die Tastatur. Sie wird immer dann aufgerufen, wenn der USART ein Zeichen empfängt. Sie wirft zunächst einige, von

Start	Größe	Beschreibung
0x60	160	Zeilenspeicher 00-03
256	240	Zeilenspeicher 04-09
496	1	line_counter wird für den Bildaufbau benötigt
497	1	time wird für delay benötigt
498	1	text_pos_x Spalte des Cursors
499	1	text_pos_y Zeile des Cursors
500	1	timer_counter zählt 4 mal pro Zeile, für äußere ISR
501	2	random_number Enthält Zufallszahl für Zufallszahlengenerator
503	1	shift_state Status der Umschalttasten
504	1	key_buffer Tastaturbuffer (1 Byte) 0 wenn leer sonst ASCII-Zeichen
512	240	Zeilenspeicher 10-15
752	16	16 Bytes frei
768	160	Zeilenspeicher 16-19
928	40	line_buffer enthält die Pixelmuster, die von outchar ausgegeben werden
968	40	line_pointers Zeiger auf alle Zeilenspeicher

Tabelle 3.1: Speicheraufbau Arbeitsspeicher

ihr nicht verwendete Zeichen weg, um dann die verbleibenden Zeichen in einer Tabelle nachzuschlagen. Dort wird jeder Taste ein Code zugewiesen, der einem Zeichen im Zeichenrom entspricht, bzw dann weiter von der Routine ausgewertet wird. Es gibt 2 Tabellen, die eine ist für den Fall, dass die Umschalttaste gedrückt ist, die andere für den Fall, dass dies nicht der Fall ist.

Das von der Tastatur eingelesene Byte wird auch noch an einen kleinen Pseudozufallszahlengenerator weitergegeben.

3.1.3 Bildunterbrechungsdienstroutine

Die deutlich kompliziertere Routine kümmert sich um den Bildaufbau. Dazu wird der 8-Bit Timer eingeschaltet, so dass er bei jedem Überlauf eine Unterbrechungsanforderung erzeugt. Diese Unterbrechungsanforderung ruft eine Unterbrechungsdienstroutine auf, die einen Zähler weiter zählt. Ist der Inhalt dieses Zählers durch 4 teilbar (ohne Rest), so wird die eigentliche Bildunterbrechungsdienstroutine aufgerufen. Diese zählt einen Zeilenzähler (line_counter) hoch und bestimmt ob die augenblickliche Zeile eine leere Zeile, die letzte Zeile, oder die Zeile ist, in der das Bild beginnt. Im ersten Fall wird einfach ein Synchronimpuls ausgegeben. Ist die Zeile die letzte Zeile, so wird ein komplette Hsync-Impuls ausgegeben und der Zähler wird wieder auf 0 zurückgesetzt. Der genaue Ablauf mit den entsprechenden Werten für line_counter findet sich in Abbildung 3.1 auf Seite 13.

Bildaufbau

Bei 16 MHz hat eine Zeile genau $\frac{16\text{ MHz}}{15{,}625\text{ kHz}}$ =1024 Taktzyklen.

Stellt die Bildunterbrechungsdienstroutine fest, dass das eigentliche Bild gezeichnet werden soll, so werden zunächst mehrere leere Zeilen gezeichnet. Diese Zeilen werden über eine Schleife gezeichnet, und sie sind somit immer genau definierte 1024 Taktzyklen lang. Die anderen, durch die Unterbrechungsdienstroutine gezeichneten Zeilen können kleine Taktungenauigkeiten aufweisen, da

line_counter	Beschreibung
0–39	leerer Bildbereich (nur horizontale Synchronimpulse)
40	Vorzeilen zum Jitterausgleich
40	Bild
41-103	leerer Bildbereich (nur horizontale Synchronimpulse)
104	vertikale Synchronimpulse

Abbildung 3.1: Bildaufbau (vertikal)

die Unterbrechungsanforderung erst nach dem gerade ausgeführten Befehl ausgeführt werden kann. Dies führt zu einer schwer vorhersagbaren Verzögerung von einigen Taktzyklen. Auf dem Bildschirm würde sich dies durch ein störendes Zittern bemerkbar machen. Die mit einer Schleife ausgegebenen Zeilen beruhigen das Signal, so dass die Ablenkung des Monitors in einem definierten Zustand ist.

Das Bild selbst wird von einer Schleife aufgebaut, die für jede Bildzeile genau einmal durchlaufen wird. An Anfang dieser Schleife wird ein Zeiger auf den Beginn der Textzeile gesetzt und ein weiterer auf den Anfang des Zeichensatzes. Ein dritter zeigt auf einen temporären Puffer.

Nun wird ein Unterprogramm aufgerufen, dass ein Zeichen der Textzeile holt, sein Muster im Zeichensatz nachschlägt und es in den temporären Puffer schreibt. Dies geschieht für jedes Zeichen der Zeile also 40 mal hintereinander. Da eine Schleife zu viele Taktzyklen benötigen würde steht diese Befehlskette in einem Makro, das einfach 40 mal aufgerufen wird.

Das wichtigere Unterprogramm ließt nun ein Byte aus dem temporären Puffer, gibt es aus, verschiebt es, gibt es aus, usw. Somit wird bei jedem 2. Taktzyklus ein Pixel ausgegeben. Eine Ausnahme ist das letzte Pixel des Zeichens. Dieses dauert 3 Taktzyklen, da ein neues Byte aus dem Speicher geladen werden muss. Das dazu verwendete Macro ist outchar. Es wird auf Seite 17 näher beschrieben. Hier sind Schleifen gänzlich unmöglich, da jeder dazu benutzte Taktzyklus sofort auf dem Bildschirm sichtbar würde.

Die restlichen ca. 30 Taktzyklen der Zeile werden in Warteschleifen verbraten. Vielleicht kann man diese für weitere Funktionen verwenden.

3.2 Programmnahe Schicht

Die programmnahe Schicht stellt Funktionen zur Verfügung, um die hardwarenahe Schicht nutzen zu können. Hier finden sich Unterprogramme die beispielsweise melden, ob eine Taste gedrückt worden ist, oder ein Zeichen auf den Bildschirm ausgeben.

3.2.1 Readline

Hier befindet sich auch eine kleine Eingaberoutine, die es ermöglicht, Texte einzugeben. Sie verfügt über eine Längenbegrenzung, sowie einfache Löschfunktionen.

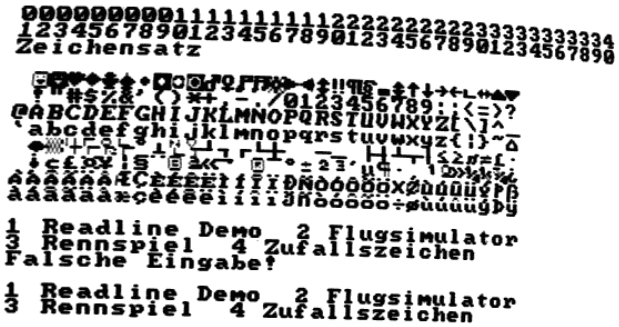

Abbildung 3.2: Bildschirmphoto des Menüs

3.3 Demoprogramm

Zur Zeit gibt es 4 Demoprogramme. Bei dem ersten wird einfach die readline-Funktion aufgerufen. Das Zweite zeigt einen Flugsimulator mit etwas spartanischer Graphik. Punkt 3 im Menü startet ein kleines Rennspiel, und Punkt 4 zeigt zufällige Zeichen auf dem Bildschirm an.

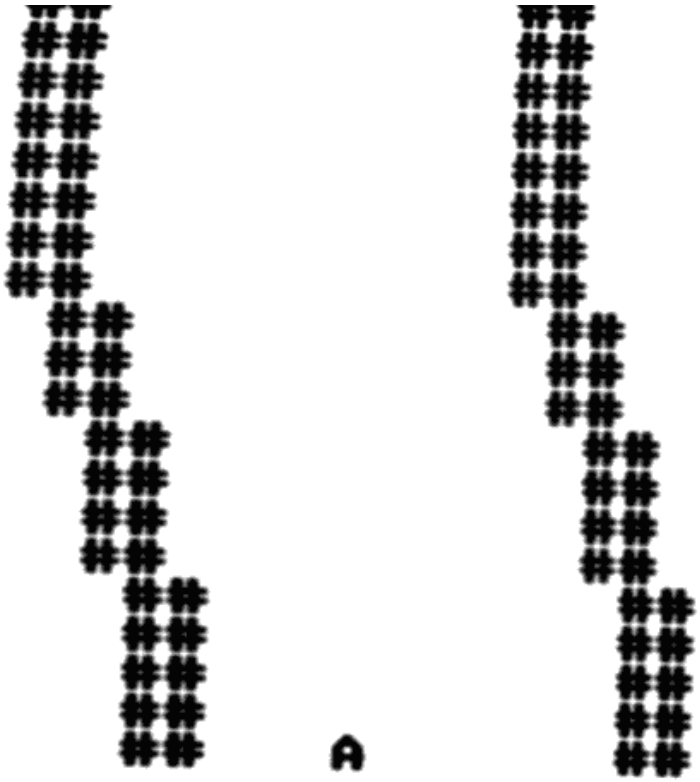

Abbildung 3.3: Rennspiel

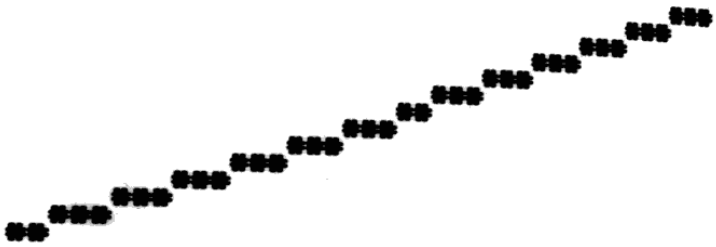

Abbildung 3.4: Flugsimulator

Abbildung 3.5: Zufallszeichen

Kapitel 4

Detailierte Dokumentation einzellner Programmteile

4.1 hardwarenah

4.1.1 Videoausgabe

outchar

outchar gibt die 8 Pixel, die sich im Byte an Adresse X befinden, die Adresse wird danach erhöht.

Taktzyklus	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Zeichen	A			B																	C
Pixel	7			0		1		2		3		4		5		6		7			0

Wie man in der Tabelle sieht dauern die meisten Pixel 2 Taktzyklen, nur Pixel 7, der letzte und somit am weitesten rechts stehende Pixel, dauert 3 Taktzyklen und ist somit etwas breiter. Im praktischen Betrieb fällt dies jedoch nicht auf.

```
.macro outchar
    LD r1,X+ ;2
    OUT video,r1 ;1 0
    LSR r1 ;1
    OUT video,r1 ;1 1
    LSR r1 ;1
    OUT video,r1        ;1 2
    LSR r1 ;1
    OUT video,r1        ;1 3
    LSR r1 ;1
    OUT video,r1        ;1 4
    LSR r1 ;1 #12
    OUT video,r1        ;1 5
    LSR r1 ;1
    OUT video,r1        ;1 6
```

```
    LSR r1 ;1
    OUT video,r1        ;1 7
                        ; 17 TZ
.endmacro
```


outchar
r1=[X]; X++
r1 ausgeben
r1 nach rechts verschieben
r1 ausgeben
r1 nach rechts verschieben
r1 ausgeben
r1 nach rechts verschieben
...
r1 nach rechts verschieben
r1 ausgeben
Ende

Kapitel 5

Ausblick

5.1 momentane Fehler

Im Moment gibt es in der Software noch ein paar kleine Fehler.

- Die Synchronimpulse sind noch nicht gleich lang. Dies sorgt für Probleme mit einigen Videokameras.
- Das Rennspiel hat noch keinen Punktezähler und keine Kollisionserkennung.

5.2 nächste Schritte

Folgendes sind die wahrscheinlichen nächsten Schritte:

5.2.1 2. Zeichensatz

Der 2. Zeichensatz kann dazu dienen, Graphikzeichen auszugeben. Dazu könnte man jedes Zeichen in 4 Pixel unterteilen, von denen jedes dann in 4 Mustern leuchten kann. Der Zeichensatz kann über ein Bit des Zeilenzeigers ausgewählt werden. Beispielsweise könnte hierfür das höchstwertige Bit verwendet werden. Das verringert zwar den adressierbaren Speicherbereich auf 32 KByte, sollte jedoch auf absehbare Zeit keine Probleme machen. Sollte dennoch einmal ein Controller mit mehr Speicher verfügbar sein, so kann man entweder die Zeiger verschoben abspeichern, so dass die niederwertigsten Bits nicht gespeichert werden, oder man verwendet einen Zeichensatz der im Arbeitsspeicher liegt.

5.2.2 Forth-Compiler

Ein Forth-Compiler würde die Möglichkeiten des Gerätes schlagartig vervielfachen. Plötzlich wäre es möglich, direkt auf dem Gerät zu entwickeln, und das in einer schnellen Hochsprache. Man würde keinen Computer mehr benötigen und könnte kleine Ideen direkt auf dem Gerät verwirklichen.

Auch könnte man den Computer dann als vollständigen Heimcomputer verwenden.

5.2.3 Selbstkopierfunktion

Eine weitere wichtige Funktion wäre die Möglichkeit, weitere Controller über die Controller selbst zu "brennen". Somit werden externe Computer überflüssig und jeder Interessierte Anwender könnte seine eigenen Controller programmieren, auch ohne einen externen Rechner benutzen zu müssen.

5.2.4 lokaler Bus

Da der SPI-Anschluss noch frei ist bietet er sich geradezu an. Dies ist ein recht schneller synchroner serieller Bus, der es ermöglicht mehrere Geräte relativ schnell mit dem Controller zu verbinden. Solche Geräte könnten beispielsweise weitere, kleinere Mikrocontroller sein, die dann beispielsweise serielle Schnittstellen zur Verfügung stellen, oder Schnittstellen zu Festplatten oder anderen Massenspeichergeräten.