Entwicklung eines „Homecomputers“ auf Basis eines AVR-Microcontrollers

Bachelorarbeit

im Studiengang Informatik

Christoph Sallie

17.09.2014

Erstprüfer: Dr. Johannes Frey

Zweitprüfer: Dr. Merten Joost

Inhaltsverzeichnis

1 Einleitung

1.1 Zielsetzung

Im Verlauf der vergangenen Jahre wurden unter der Leitung von Dr. Merten Joost basierend auf Microcontrollern der ATMega-Reihe verschiedene Projekte zur Ansteuerung der Peripheriegeräte eines Computers realisiert [1]. Hierzu zählen unter anderem die Abfrage einer Tastatur, die Ausgabe von Audio- und Videosignalen sowie eine Programmierumgebung mit eigener Programmiersprache.
Ziel dieser Arbeit ist es, die gesammelten Projekte zu verbinden, um als Ergebnis einen eigenständigen „Homecomputer“ zu erhalten, der per Tastatur angesteuert werden können soll und über eine Audio- und Videoausgabe verfügen soll.
Dabei wird eine SD-Karte als Speichermedium dienen, das per Tastatureingabe über eine Art Shell verwaltet werden kann.

1.2 Vorhandene Arbeiten

Als Grundlage für diese Arbeit dient die von Simeon Maxim [2] entwickelte Programmierumgebung für die ATMega-Reihe. Diese bietet mit einer selbst entwickelten Programmiersprache namens „Femto“ alle Funktionen, die notwendig sind, um eigene Programme auf dem Microcontroller zu entwickeln.
Seine Arbeit beinhaltete bisher eine Schwarz-Weiß-Ausgabe und soll nun um die Farbkomponente erweitert werden.
Die Video- und Audioausgabe basiert auf weiteren Projekten, welche unter der Leitung von Dr. Merten Joost entwickelt wurden. [1]

2 Hardware

Das Zentrum der Platine bildet ein ATMega-1284P. Er bietet mit 128 KB Flash- sowie 4 KB EEProm und 16K Bytes SRam genug Platz für die umfangreichen Routinen zur Verwaltung der Peripheriegeräte.
Für die weiteren Anschlüsse wurden außerdem eine 15-polige Mini-D-Sub-Buchse für die Videoausgabe, eine 3,5 mm-Stereobuchse, ein PS/2-Anschluss und ein Micro-SD-Kartenadapter platziert.
Die Stromversorgung erfolgt über einen USB-Stecker.
Im Folgenden werden der Aufbau der Platine und die relevanten Informationen zur Kommunikation über die verschiedenen Schnittstellen genauer beschrieben.

2.1 Die PS/2-Schnittstelle

Die PS/2-Schnittstelle dient im Allgemeinen der Kommunikation zwischen Tastatur bzw. Maus und einem PC und war lange Zeit der Standard zum Anschluss dieser Geräte. Inzwischen wurde der Anschlusstyp von dem USB-Port abgelöst.
Aufbau und Arbeitsweise der Schnittstelle wird im Weiteren nur grob beschrieben, da der Teil bereits in früheren Projekten ausgiebig bearbeitet wurde.
Für nähere Informationen zu den Funktionen, Initialisierung und Kommunikation bietet sich die Arbeit von Simeon Maxim [2] zur Entwicklung einer Programmierumgebung für den ATMega an, da diese auch als Vorlage für die vorliegende Arbeit verwendet wurde.

2.1.1 Scancodes

Zur Übermittlung der Daten von Tastatur zu PC werden Scancodes verwendet. Diese bestehen aus den sogenannten Make- und Breakcodes. Eine solche Unterscheidung ist notwendig, um dem PC zu übermitteln ob eine Taste gedrückt oder losgelassen wurde. Wird eine Taste verwendet, sendet die Tastatur beim Betätigen den jeweiligen Makecode an den Computer und den dazugehörigen Breakcode, sobald die Ausgangsposition wieder erreicht

wurde.Dieser entspricht in den meisten Fällen dem Makecode, dem lediglich die Bytefolge 0xF0 vorangestellt wird.

Insgesamt gibt es drei verschiedene Sammlungen von Scancodes, sogenannte Scancode-Sets. Die Standardsammlung und auch diejenige, die bei diesem Projekt verwendet wird, ist das Scancode-Set 2.

Abb. 2.1 zeigt einen kurzen Auszug der Codes, die komplette Liste ist im Anhang zu finden.

Taste	Makecode	Breakcode
A	0x1C	0xF0, 0x1C
Enter	0x5A	0xF0, 0x5A
Pfeil nach Rechts	0xE0, 0x74	0xE0, 0xF0, 0x74

Abbildung 2-1: Beispiel Scancodes [5]

Abgesehen von den Scancodes gibt es noch eine Reihe von Befehlen, die der Computer an die Tastatur senden kann, um bestimmte Informationen zu erhalten oder gewünschte Aktionen zu erzwingen. Da diese Funktionen für die Arbeit jedoch nicht erheblich sind, wird im Folgenden nicht näher auf sie eingegangen.

2.1.2 Allgemeine Beschreibung der Schnittstelle

Der PS/2-Stecker setzt sich aus den Pins für Spannung, Erde, Clock- und Datenleitung zusammen. Außerdem existieren zwei Pins, die nicht belegt sind.

Im Folgenden sind der Aufbau und der Zweck der einzelnen Anschlüsse aufgeführt.

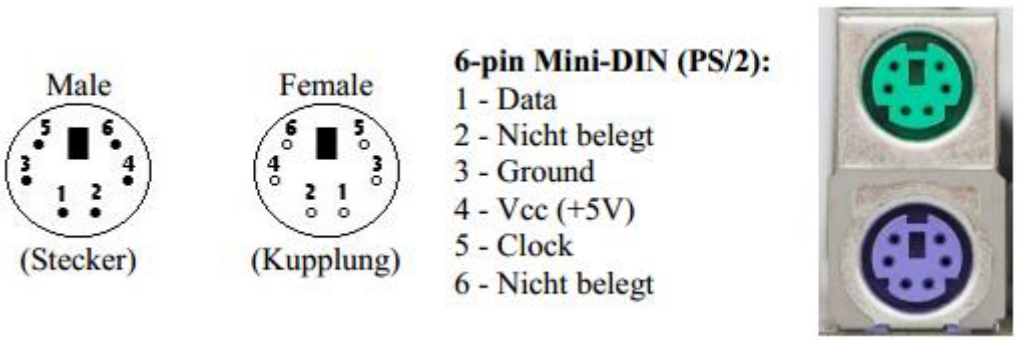


Abbildung 2-2: Belegung des PS/2-Anschluss [5]

VCC/Ground:

Über diese Pins wird das Eingabegerät mit Strom versorgt. Die Tastatur benötig eine Spannung zwischen +4.5V und 5.5V.

Clock:

Die Clock dient der Synchronisation der Signale von Host und Eingabegerät. Der Takt wird immer von der Tastatur generiert, sodass hier kein zusätzlicher Aufwand notwendig ist.

Data:

Die Datenleitung dient der Übermittlung von Daten. Die einzelnen Bits werden gelesen, wenn der Takt einen Low-Pegel liefert.

Clock- und Datenleitung sind beide bidirektional. Für beide gilt +5V entspricht einer logischen 1 und 0V entspricht der logischen 0.

2.1.3 Die Kommunikation

Im Normalfall registriert sich die Tastatur an dem PC, um die Clockfrequenz, das Scancode-Set und weitere Informationen mitzuteilen, die zum richtigen Lesen der gesendeten Signale notwendig sind. Weil die Initialisierung jedoch für die Tastatur nicht notwendig ist und sie auch ohne Weiteres direkt nach Anschluss verwendet werden kann, wird dieser Ablauf nicht näher erläutert, sondern es wird direkt auf das Protokoll zur Datenübermittlung zwischen Host und Controller eingegangen.

Die Kommunikation findet immer Byteweise statt und besteht aus den folgenden Bits:

- 1 Startbit, das immer 0 ist
- 8 Datenbits
- 1 Paritäts-Bit (1 bei gerader Anzahl an Einsen, sonst 0)
- 1 Stoppbit, das immer 1 ist
- 1 Acknowledge-Bit (wird nur bei der Kommunikation von Host zur Tastatur gesendet)

Damit die Tastatur Daten senden kann, muss diese erst überprüfen, ob die Clockleitung frei ist, d. h. sie muss auf High-Pegel sein. Ist das nicht der Fall, blockiert der Host im Moment die

Leitung und der Sendevorgang kann erst starten, wenn die Freigabe durch den PC bzw. den Microcontroller erfolgt.
Ist ein entsprechendes Signal vorhanden, fängt die Tastatur mit dem Sendevorgang an, indem sie den Takt ausgibt und immer bei High-Pegel das aktuelle Bit an den Host sendet. Dieser liest das Signal, sobald an der Clockleitung wieder ein Low-Pegel anliegt. So werden die 11 Bits übertragen, mit dem letzten Clocksignal wird die Übertragung beendet und die Tastatur gibt anschließend beide Leitungen wieder frei.

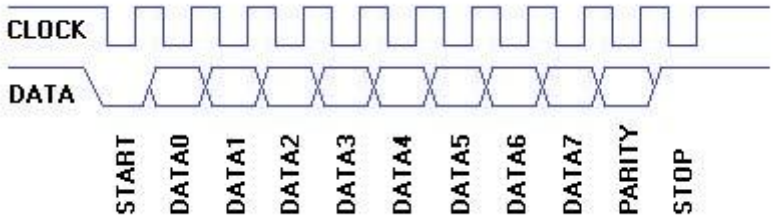


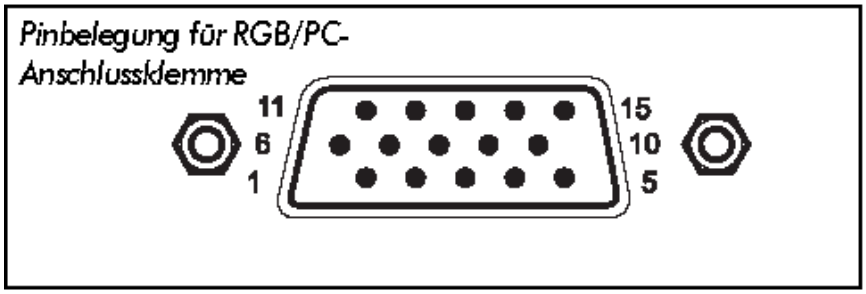
Abbildung 2-3: Kommunikation Tastatur - Host [5]

2.2 Der VGA-Anschluss

VGA steht für „Video Graphics Array“. Dies bezeichnet einen Grafikkartenstandard, über den Monitore an verschiedene Endgeräte angeschlossen werden können. Dieser wurde 1987 von IBM eingeführt und ist ein Nachfolger von EGA und CGA.
Zwar haben sich die VGA-Anschlüsse lange gehalten, werden aber inzwischen immer häufiger von anderen digitalen Schnittstellen wie HDMI verdrängt.

2.2.1 Allgemeine Beschreibung der VGA-Schnittstelle

Als VGA-Anschluss wird ein 15-poliger Mini-D-Sub-Stecker bezeichnet. Die Übertragung von Grafikkarte zu Bildschirm erfolgt über sechs Leitungen.
Die einzelnen Pins sind mit je einem Kanal für die Farbkomponenten Rot, Gelb und Blau sowie je einem Pin für das horizontale und vertikale Synchronisationssignal belegt.
Vier weitere Pins sind mit der Masse verbunden, die restlichen können optional an weitere Ausgänge angeschlossen werden, sind aber hier nicht weiter von Bedeutung.



Pin-Nr.	Signalname	Pin-Nr.	Signalname
1	R (P_R*)	9	NA
2	G (Y*)	10	Masse
3	B (P_B*)	11	NA
4	NA (nicht angeschlossen)	12	NA
5	NA	13	H-Sync
6	Masse	14	V-Sync
7	Masse	15	NA
8	Masse		

Abbildung 2-4: Pinbelegung VGA-Schnittstelle [6]

2.2.2 Synchronisation

Wie bereits erwähnt, gibt es im Fall der VGA-Schnittstelle zwei Synchronisationssignale, jeweil eins für die Horizontale und die Vertikale. An beiden liegt im Normalfall ein High-Pegel an und bei fallender Flanke findet die Synchronisation statt.

Das horizontale Signal teilt dem Bildschirm mit, wann ein Zeilenende erreicht ist und der Sprung zur nächsten notwendig wird. Da dieser Übergang etwas Zeit benötigt, werden am Ende und am Anfang jeder Zeile weitere Pixel eingefügt, die auf dem eigentlichen Bild nicht mehr zu sehen sind.

Die vertikale Synchronisation übermittelt Anfang und Ende eines Bildes an den Bildschirm. Da auch hier eine gewisse Zeit benötigt wird, um den Sprung von dem Ende bis zum Bildschirmanfang zu überwinden, werden wieder zusätzliche Zeilen übermittelt, die in dem Ergebnis nicht mehr zu sehen sind.

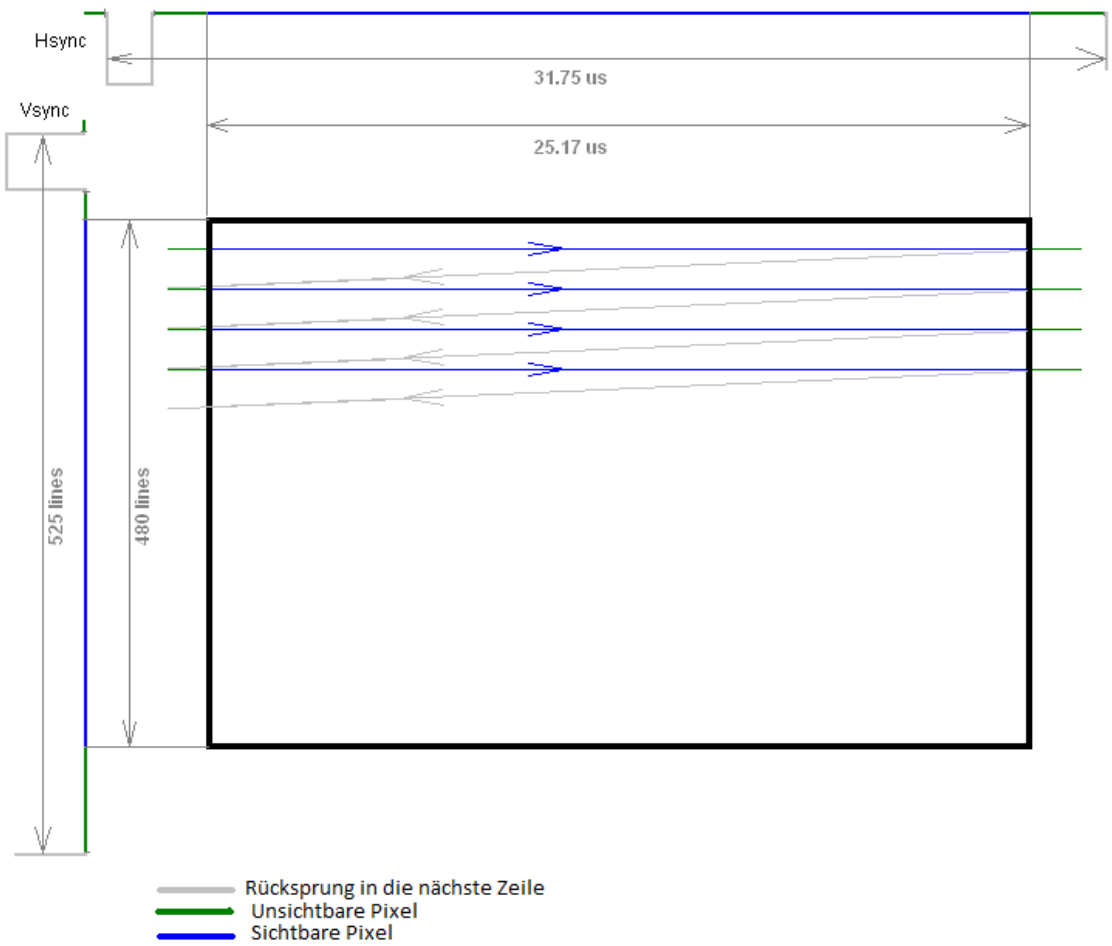


Abbildung 2-5: Verlauf eines Elektronenstrahls bei Röhrenbildschirmen

2.2.3 Die Darstellung

In der Zeit, in der die sichtbaren Zeichen übermittelt werden, liegen an den drei Farbkanälen analoge Signale an. Diese sind relativ zu der Spannung am Anfang und Ende jeder Zeile. D. h. das Signal, das während der nicht sichtbaren Bereiche angelegt war, gibt den Pegel für Schwarz bzw. den dunkelsten Ton der entsprechenden Farbe an.

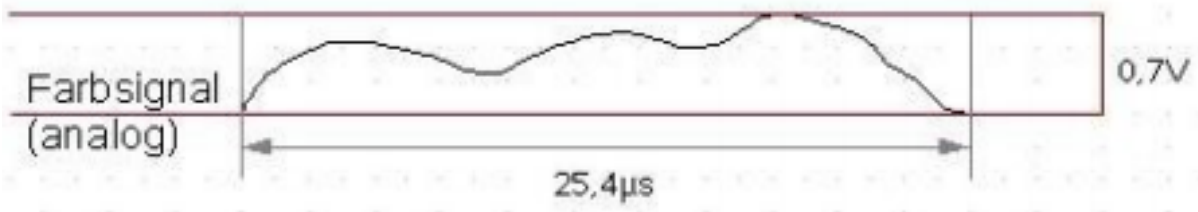


Abbildung 2-6: analoges Farbsignal [1]

Bewegt sich dieser um 0,7 V nach oben, erhält man den jeweiligen hellsten Farbton. In den Bereichen dazwischen kann beliebig variiert werden.

Die übertragenen Daten werden bei Röhrenmonitoren durch einen Elektronenstrahl dargestellt, der zeilenweise über den Bildschirm wandert. Die Stärke des Strahls richtet sich nach der anliegenden Spannung.

Auf dem Bildschirm wird die resultierende Farbe der drei Kanäle dann durch additive Farbmischung erzeugt.

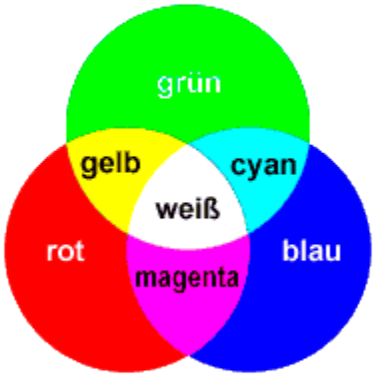
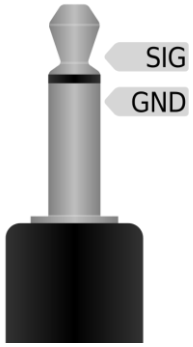


Abbildung 2-7: Additive Farbdarstellung [1]

2.3 Die Audioausgabe

Der Standard zum Anschluss von Audiogeräten wird „Sleeve“ oder „Klinke“ genannt. Es gibt ihn in verschiedenen Formen und Größen.
Von der kleinsten Variante – einem zweipoligem Monostecker (2,5 mm), der zwei Signalflächen besitzt, eine als Toneingang und eine als Masseverbindung (siehe Abbildung) – bis hin zum sogenannten „NATO-Plug“ (7,13 mm), der allerdings nur in der militärischen Luftfahrt verwendet wird.



2-8 zweipoliger Monostecker [7]

Auf der in dieser Arbeit behandelten Platine wird ein dreipoliger Stereostecker (3,5 mm) verwendet, den man auch in den meisten tragbaren Geräten wie MP3-Playern und an PCs findet. Dieser hat einen Anschluss für die Masse sowie jeweils eine Verbindung für linkes und rechtes Tonsignal.

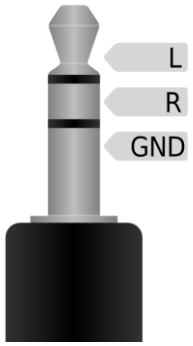


Abbildung 2-9: dreipoliger Stereostecker [7]

Das Erzeugen eines Audiosignals erweist sich als vergleichsweise einfach. Das, was wir im Allgemeinen als Ton wahrnehmen, bezeichnet eigentlich die Ausbreitung von Schall. Dieser ist physikalisch gesehen eine Welle, die sich durch ihre Frequenz und die Amplitude auszeichnet. Diese beiden Faktoren werden als Tonhöhe und Frequenz wargenommen.

Ton	Frequenz
C4	262 Hz
C4#	277 Hz
D4	294 Hz
E4	330 Hz

Ton	Frequenz
F4	349 Hz
F4#	370 Hz
G4	392 Hz
G4#	415 Hz

Ton	Frequenz
A4	440 Hz
A4#	466 Hz
B4	494 Hz
C5	523 Hz

Abbildung 2-10: Töne und ihre Frequenzen [1]

Da Kopfhörer und Boxen so aufgebaut sind, dass die Spannung und Frequenz eines anliegenden analogen Signals direkt in den entsprechenden Ton umgewandelt und verstärkt wird, reicht es aus, ein entsprechendes wellenförmiges Signal ausgeben.
Dieses lässt sich durch die intergierte Pulsweitenmodulation (PWM) der Timer eines ATMega leicht ausgeben.

2.4 Die SD-Karte

Die SD-Karte ist ein Flash-Speichermedium, das 2001 auf Basis des MMC-Standards entwickelt wurde. Im Gegensatz zu den älteren Karten verfügt sie über vier statt nur eine Datenleitung und ermöglicht so eine schnellere Datenübertragung. Für den Anschluss an den ATMega werden hier allerdings nur zwei dieser Leitungen verwendet.
In diesem Projekt wird eine Micro-SD-Karte verwendet, die allerdings im Aufbau der Standardkarte entspricht.
Da die Karte im Gegensatz zu dem ATmega mit einer Spannung von 2,7V-3,6V arbeitet, wird vor jedem Eingangspin der SD-Karte ein Spannungsteiler platziert.

2.4.1 Aufbau und Anschluss der SD-Karte

Die meisten SD-Karten beinhalten zur Ansteuerung neben dem normalen Protokoll auch einen SPI-Modus. Daher bietet es sich an, den entsprechenden Anschluss des ATMega zu nutzen.
Abbildung 2-10 zeigt zusammen mit Tabelle 1 die Pinbelegung der SD-Karte.
Für die Kontakte MOSI, MISO und SS gilt, dass sie mit den gleichnamigen Pins des Microcontrollers verbunden werden müssen.
Der SS-Kontakt (Slave-Select) kann wiederrum an einen beliebigen anderenn Pin des ATMega angeschlossen werden. Mit dieser Leitung wird dem Slave signalisiert, dass er Daten empfangen soll, indem die Leitung auf Low-Pegel gezogen wird.

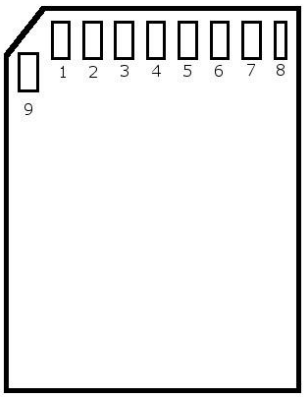


Abbildung 2-11: Pinbelegung SD-Karte

Pin	Name	Funktion (SD-Modus)	Funktion (SPI-Modus)
1	DAT3/CS	Data Line 3	Chip Select/Slave Select (CS / SS)
2	CMD/DI	Command Line	Master Out Slave In (MOSI)
3	GND1	Ground	Ground
4	VDD	Supply Voltage	Supply Voltage
5	CLK	Clock	Clock (SCK)
6	GND2	Ground	Ground
7	DAT0/DO	Data Line 0	Master In Slave Out (MISO)
8	DAT1/IRQ	Data Line 1	Unused or IRQ
9	DAT2/NC	Data Line 2	Unused

Tabelle 1: SD-Karten-Pins-Zuordnung

2.4.2 Der SPI-Modus

Der SPI-Modus ist eine alternative Methode zum Ansteuern von MMC-/SD-Karten. Im Gegensatz zu dem Standardverfahren, ist das Kommunikationsprotokoll für den SPI-Modus ein wenig simpler gehalten. Daher bietet es sich an, ihn für die Ansteuerung durch einen Microcontroller zu verwenden.

Es gibt vier verschiedene Versionen des SPI-Modus, die beschreiben, wann die Bits gelesen und übertragen werden.

Zur Ansteuerung einer SD-Karte wird der Modus 0 verwendet, d. h. die Lese- und Schreibvorgänge werden bei steigender Flanke der Clock durchgeführt.

2.4.3 Kommando und Antwort

Die Kommunikation zwischen Host und SD-Karte verläuft nach dem Frage-Antwort-Prinzip: Benötigt der Host eine Information, wird über die CMD-Leitung ein Kommando an die Karte gesendet und anschließend auf die Antwort gewartet.

Die Kommandos bestehen immer aus einem Paket mit einer festgesetzten Länge, die byteweise an die Karte gesendet werden.

Zu Beginn der Datenübertragung wird die SS-Leitung auf Low-Pegel gezogen um der SD-Karte zu signalisieren, dass ein Kommando gesendet wird. Diese ist bereit, Daten zu empfangen sobald sie die MISO-Leitung auf High-Pegel setzt.

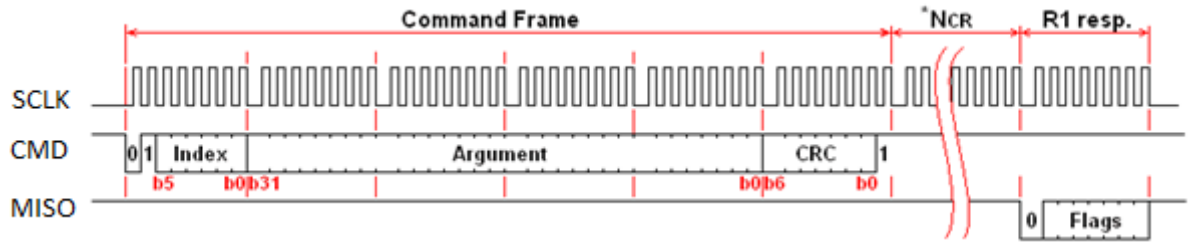


Abbildung 2-12: Kommunikation Host - SD-Karte [8]

Ist das Kommando übermittelt, muss der Host weiter das Clock-Signal generieren und über die MOSI-Leitung dauerhaft 0xFF senden, bis die erwartete Antwort der SD-Karte erhalten wurde, was zwischen 0 und 8 Bytes dauert. Erfolgt in dieser Zeit keine valide Antwort, war die Datenübertragung fehlerhaft.

Jedes dieser Kommandos besteht aus einem Kommandoindex, ggf. dem Argument und einem CRC-Feld. Letzteres ist jedoch im SPI-Modus optional und wird von der Karte nicht überprüft.

Der Kommandoindex wird immer bezeichnet mit CMD(n), wobei (n) jeweils eine Zahl zwischen 0 und 63 ist und dem Integerwert entspricht, der an die SD-Karte übermittelt wird. Tabelle 2 zeigt einen Auszug der für Lese-/Schreibzugriff und die Karteninitialisierung notwendigen Befehle.

Command Index	Argument	Response	Data	Abbreviation	Description
CMD0	None (0)	R1	No	GO_IDLE_STATE	Software reset
CMD1	None (0)	R1	No	SEND_OP_COND	Initiate intialization Process
ACMD41	*2	R1	No	APP_SEND_OP_COND	For only SDC. Initiate initialization process
CMD8	*3	R7	No	SEND_IF_COND	For only SDC V2. Check voltage range
CMD9	None (0)	R1	Yes	SEND_CSD	Read CSD register
CMD10	None (0)	R1	Yes	SEND_CID	Read CID register
CMD12	None (0)	R1	No	STOP_TRANSMISSION	Stop to read data
CMD16	Block length[31:0]	R1b	No	SET_BLOCKLEN	Change R/W block size
CMD17	Address[31:0]	R1	Yes	READ_SINGLE_BLOCK	Read a block
CMD18	Address[31:0]	R1	Yes	READ_MULTIPLE_BLOCK	Read multiple blocks
CMD23	Number of blocks [15:0]	R1	No	SET_BLOCK_COUNT	For only MMC. Define number of blocks to transfer with next multi-block read/write command
ACMD23	Number of blocks [22:0]	R1	No	SET_WR_BLOCK_ERASE_COUNT	For only SDC. Define number of blocks to pre-erase with next multi-block write command
CMD24	Address [31:0]	R1	Yes	WRITE_BLOCK	Write a block
CMD25	Address[31:0]	R1	Yes	WRITE_MULTIPLE_BLOCK	Write multiple blocks
CMD55	None (0)	R1	No	APP_CMD	Leading command of ACMD(n) command
CMD58	None (0)	R3	No	READ_OCR	Read OCR

*1 ACMD(n) means a command sequence of CMD55-CMD(n)

*2: Rsv(0) [31], HCS[30], Rsv(0)[29:0]

*3: Rsv(0) [31:12], Supply Voltage(1) [11:8], Check Pattern (0xAA) [7:0]

Tabelle 2: Kommandos an die SD-Karte [8]

Zu jedem Kommando wird im SPI-Modus auch eine Antwort (R1, R2, R3, R7) des Microcontrollers erwartet. Im Falle einer erfolgreichen Übertragung hat die Antwort den Wert 0x00. Ist hingegen ein Fehler aufgetreten, wird das entsprechende Bit gesetzt.
Die Antworten R3 und R7 bestehen aus R1 gefolgt von einem 32-bit Datenblock und sind für die Kommandos 58 und 8 reserviert.
Da manche Befehle etwas mehr Zeit zur Ausführung beanspruchen, wird in dem Fall die Antwort R1b gesendet. Sie entspricht R1, jedoch wird hier zusätzlich ein Flag angehängt, das signalisiert, dass die Karte momentan noch beschäftigt ist.
In den nachfolgenden drei Tabellen wird der Aufbau der einzelnen Antworttypen und deren Bedeutung kurz erklärt.

Byte	Bit	Bedeutung
1	7	Start Bit, immer 0
	6	Parameter Error
	5	Address Error
	4	Erase Sequence Error
	3	CRC Error
	2	Illegal Command
	1	Erase Reset
	0	In Idle State

Tabelle 3: Antworttyp R1

Byte	Bit	Bedeutung
1	7	Start Bit, immer 0
	6	Parameter Error
	5	Address Error
	4	Erase Sequence Error
	3	CRC Error
	2	Illegal Command
	1	Erase Reset
	0	In Idle State
2	7	Out of Range, CSD Overwrite
	6	Erase Parameter
	5	Write Protect Violation
	4	Card ECC Failed
	3	Card Controller Error
	2	Unspecified Error
	1	Write Protect Erase Skip, Lock/Unlock Failed
	0	Card Locked

Tabelle 4: Antworttyp R2

Byte	Bit	Bedeutung
1	7	Start Bit, immer 0
	6	Parameter Error
	5	Address Error
	4	Erase Sequence Error
	3	CRC Error
	2	Illegal Command
	1	Erase Reset
	0	In Idle State
2-5	All	Operation condition Register, MSB First

Tabelle 5: Antworttyp R3

2.4.4 Initialisierung

Sobald die SD-Karte mit Strom versorgt wird, startet sie im Standard.odus. Um nun den SPI-Modus zu aktivieren, muss eine bestimmte Initialisierung durchgeführt werden. Dieser Vorgang wird im Folgenden weiter beschrieben. Abbildung 2-10 zeigt diesen Prozess in einem Fluss-Diagramm.

Power ON

Nachdem die Spannung 2,2 Volt erreicht hat, wird eine Millisekunde gewartet und die Taktrate auf 100 bis 400 kHz gesetzt.
Nun werden mit einem High-Pegel auf der CMD- und SS-Leitung 74 oder mehr Clock-Impulse gesendet, da die Karte etwas Zeit braucht, um für den Datenempfang bereit zu sein.

Software Reset

Anschließend wird die SS-Leitung auf Low-Pegel gezogen und CMD0 an die Karte gesendet, um sie zurückzusetzen.
Wurde das Kommando erfolgreich von der Karte empfangen, antwortet sie mit R1 und gesetztem „In Idle State“-Bit (0x01).

Initialisierung

Befindet sich die Karte im „Idle State“, akzeptiert sie lediglich die Kommandos CMD0, CMD1, ACMD41, CMD58, und CMD59. Alle anderen werden abgelehnt.
Zunächst wird nun die Versorgungsspannung überprüft, die sich im Bereich von 2,7 bis 3,6 Volt befinden sollte.
Sobald die Karte das Kommando CMD1 erhält, startet sie die Initialisierung. Wurde diese beendet, erhält der Microcontroller die Antwort R1 (0x00).
Für SD-Karten ist ACMD41 empfohlen, daher empfiehlt es sich, zuerst dieses Kommando zu senden. Wird es abgelehnt, wird ein neuer Versuch mit CMD1 gestartet.
Anschließend kann die Taktrate auf den maximal möglichen Wert gesetzt werden. Im Normalfall liegt dieser zwischen 20 und 25 MHz.

Unterstützung von SD-Karten Version 2

Damit auch SD-Karten Version 2 (im Allgemeinen Karten mit einer Speicherkapazität > 2GB) unterstützt werden können, müssen bei der Initialisierung einige Dinge hinzugefügt werden.
Sobald sich die Karte im „Idle State“ befindet, wird CMD8 mit dem Argument 0x000001AA gesendet. Wird dieses von der Karte abgelehnt, handelt es sich um eine SD-Karte Version 1. Andernfalls erhält man die Antwort R7. Hier haben die letzten 12 Bits den Wert 0x1AA und sagen aus, dass es sich um eine SD-Karte Version 2 handelt und sie in dem geforderten Spannungsbereich arbeitet.

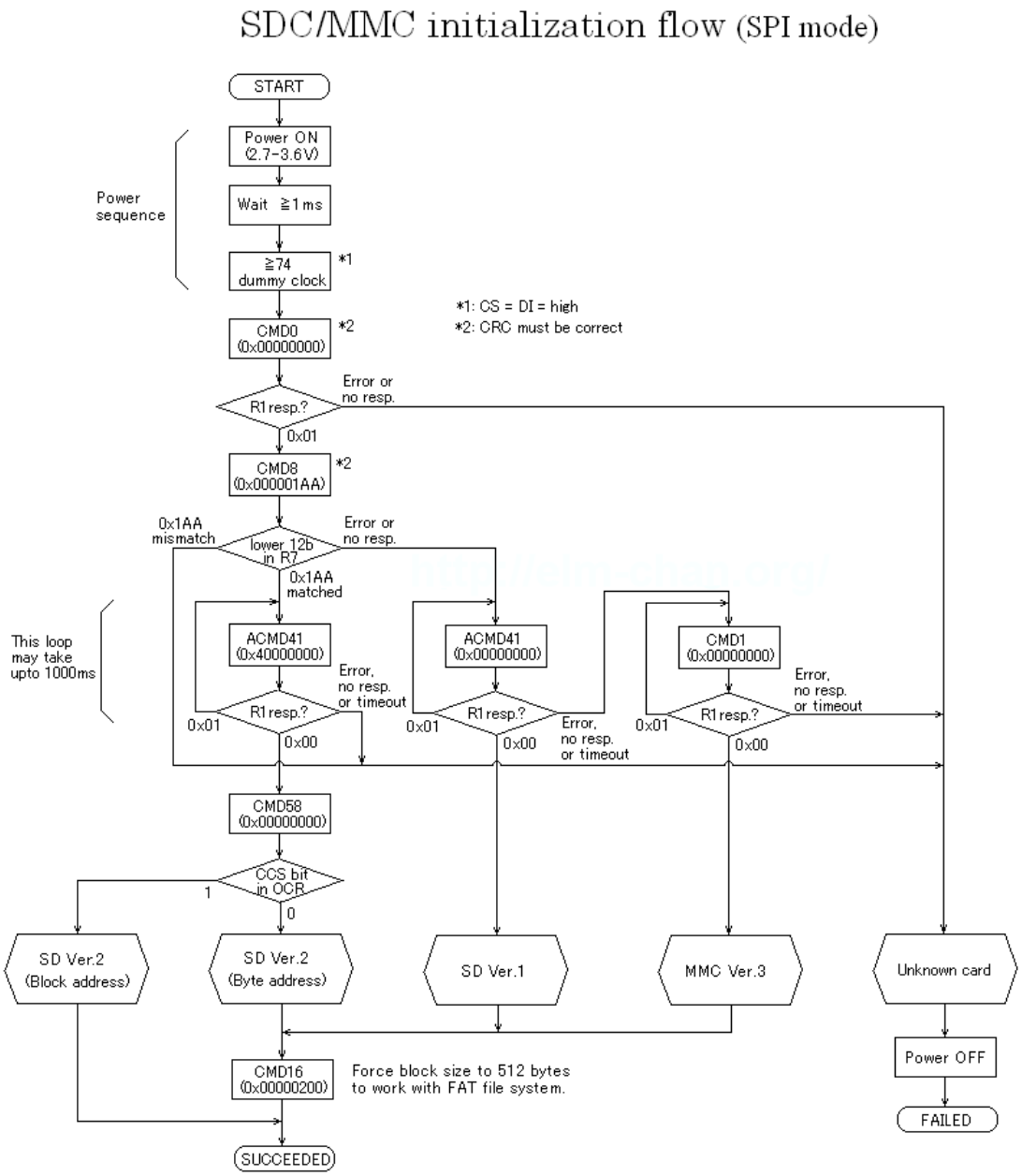


Abbildung 2-13: Initialisierung SD-Karte [8]

2.4.5 Die Datenübertragung

Bei der Übertragung im Lese-/Schreibprozess werden die Daten im Anschluss auf die Antwort des Microcontrollers gesendet bzw. erhalten. Diese Daten werden immer in Datenblöcken bestimmter Größe übertragen, die sich zwar bei vielen Karten ändern lässt, jedoch in unserem Fall bei dem Standard von 512 Bytes pro Block belassen wird.

Einzelnen Datenblock lesen

Zum Lesen eines einzelnen Datenblocks von der SD-Karte sendet der Microcontroller das Kommando CMD17 mit der Startadresse des zu lesenden Bereichs als Argument. Nach Erhalt einer validen Antwort werden die entsprechenden Bytes an den Host übermittelt.

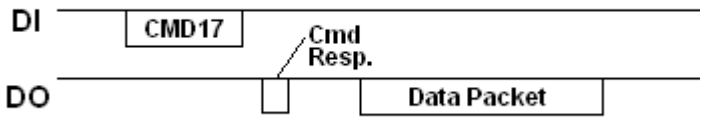


Abbildung 2-14: Lesen eines einzelnen Datenblocks [8]

Mehrere Datenblöcke lesen

Um mehrere Datenblöcke zu lesen, wird das Kommando CMD18 verwendet. Dieses startet einen kontinuierlichen Lesevorgang. Wurde im Vorfeld die Anzahl der zu lesenden Blöcke durch das Kommando CMD23 definiert, endet die Übertragung nach dieser Zeit.
Ist dies nicht der Fall, werden weiter Daten gesendet, bis der Lesevorgang mit dem Kommando CMD12 abgebrochen wird.

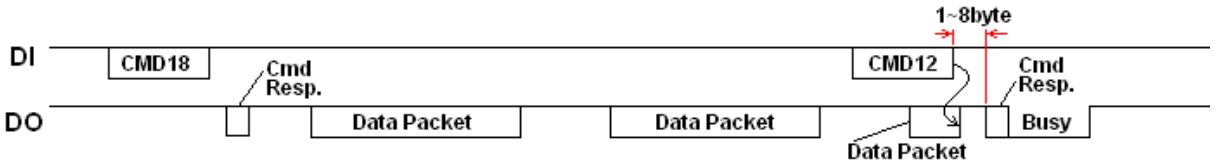


Abbildung 2-15: Lesen mehrerer Datenblöcke [8]

Einzelnen Datenblock schreiben

Um einen einzelnen Block auf die SD-Karte zu schreiben, wird das Kommando CMD24 verwendet. Wurde dieses Kommando akzeptiert, sendet der Host ein Datenpaket. Anschließend übermittelt die Karte eine Antwort und ein Flag, um zu signalisieren, dass sie noch mit dem Schreibvorgang beschäftigt ist. Daher muss nach jedem Datenpaket eine Überprüfung stattfinden, ob die Karte wieder bereit ist, bevor ein weiteres Kommando oder Paket gesendet wird.

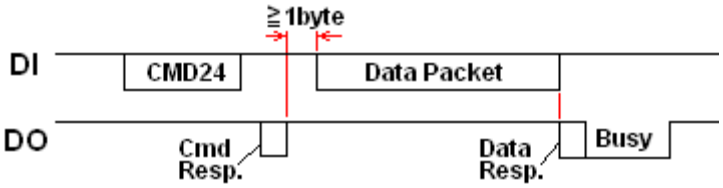


Abbildung 2-16: Schreiben eines einzelnen Datenblocks [8]

Mehrere Datenblöcke schreiben

Um das Schreiben einer größeren Anzahl von Datenblöcken zu initiieren, wird das Kommando CMD25 verwendet.

Wie auch beim Lesen wird hier kontinuierlich geschrieben, bis entweder ein Abbruchkommando oder die über CMD23 vordefinierte Menge an Blöcken erreicht wurde.

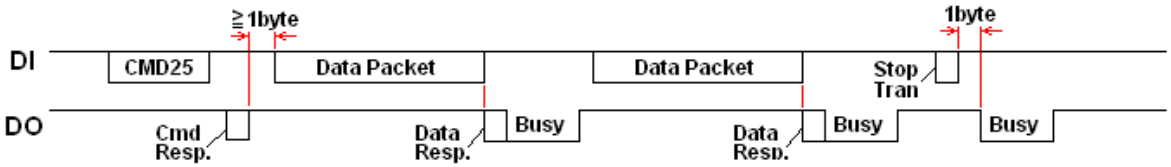


Abbildung 2-17: Schreiben mehrerer Datenblöcke [8]

2.5 Das Board

Das Zentrum des Boards bildet ein Atmel ATMega1284p. Hier werden alle ein- und ausgehenden Daten der diversen Peripheriegeräte verwaltet.
Die einzelnen Bestandteile werden im Folgenden genauer erklärt.

Die Stromversorgung:

Zur Stromversorgung wird ein USB-Anschluss verwendet. Da dieser bereits eine +5V-Spannung liefert, sind an dieser Stelle keine weiteren Bauteile nötig.
Zusätzlich wurde eine LED als Kontrollleuchte eingefügt.

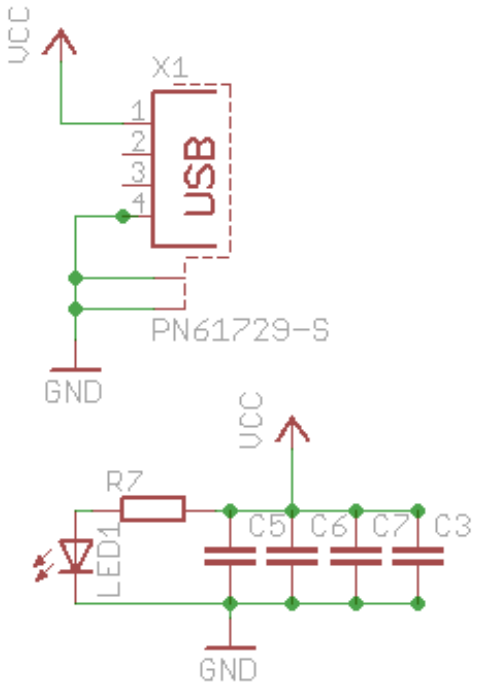


Abbildung 2-2-18: Die Stromversorgung

Die PS/2-Schnittstelle:

Die PS/2-Schnittstelle benötigt lediglich zwei Pins des Microcontrollers. Die Clockleitung wird mit Pin B0 und die Datenleitung mit Pin D0 verbunden.
Außerdem besitzt sie je einen Anschluss für Masse und Spannungsversorgung.

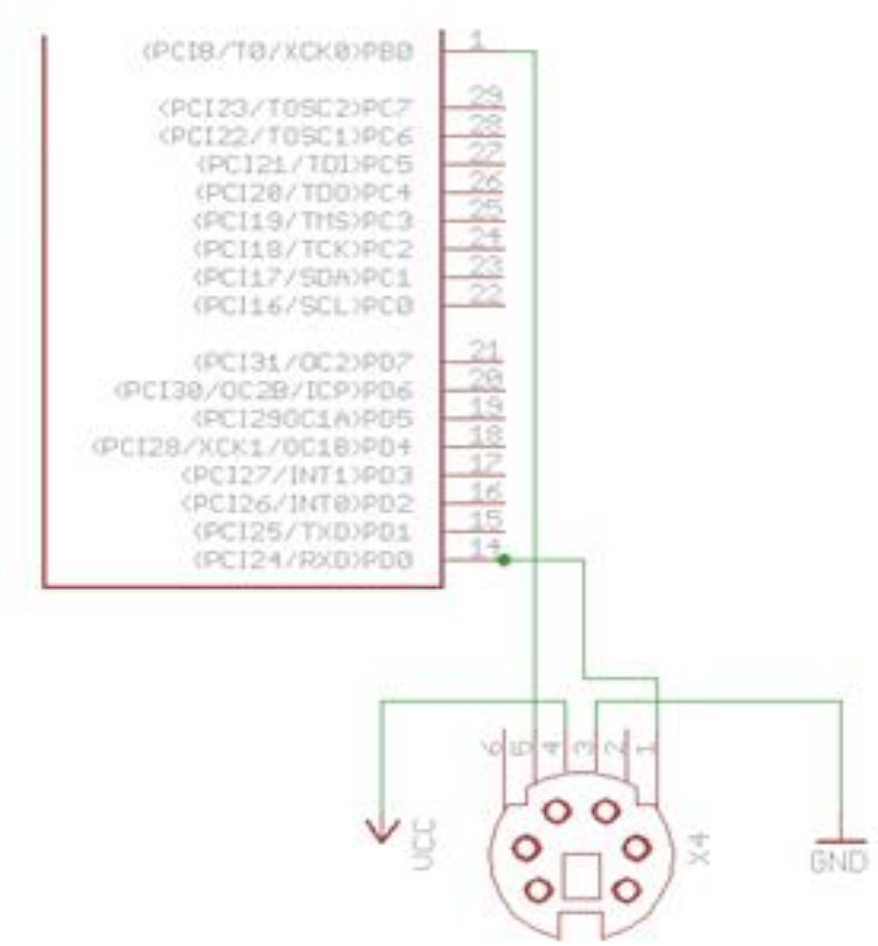


Abbildung 2-2-19: Die PS/2-Schnittstelle

Die Audiobuchse:

Da der Microcontroller das Signal direkt über die Timer erzeugt, kann die Audiobuchse direkt an die entsprechenden Pins (D5 und D7) angeschlossen werden.

Darüber hinaus wird eine Verbindung zur Masse benötigt.

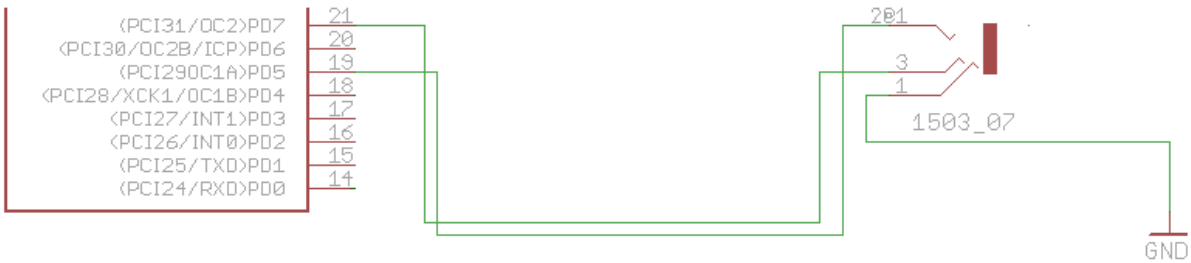


Abbildung 2-2-20: Die Audiobuchse

Die SD-Karte:

Wie schon im Abschnitt über die SD-Karte beschrieben, werden die Pins für die SPI-Schnittstelle verwendet. Diese sind PIN B8, PIN B7, PIN B6 für die MOSI, MISO und Clock sowie ein beliebiger für die SS-Leitung. In diesem Fall wurd dafür PIN B3 verwendet.
Vor jedem dieser Pins wurde ein Spannungsteiler eingebaut, um die Eingangsspannung von 2,6 - 3,3 V zu erhalten.

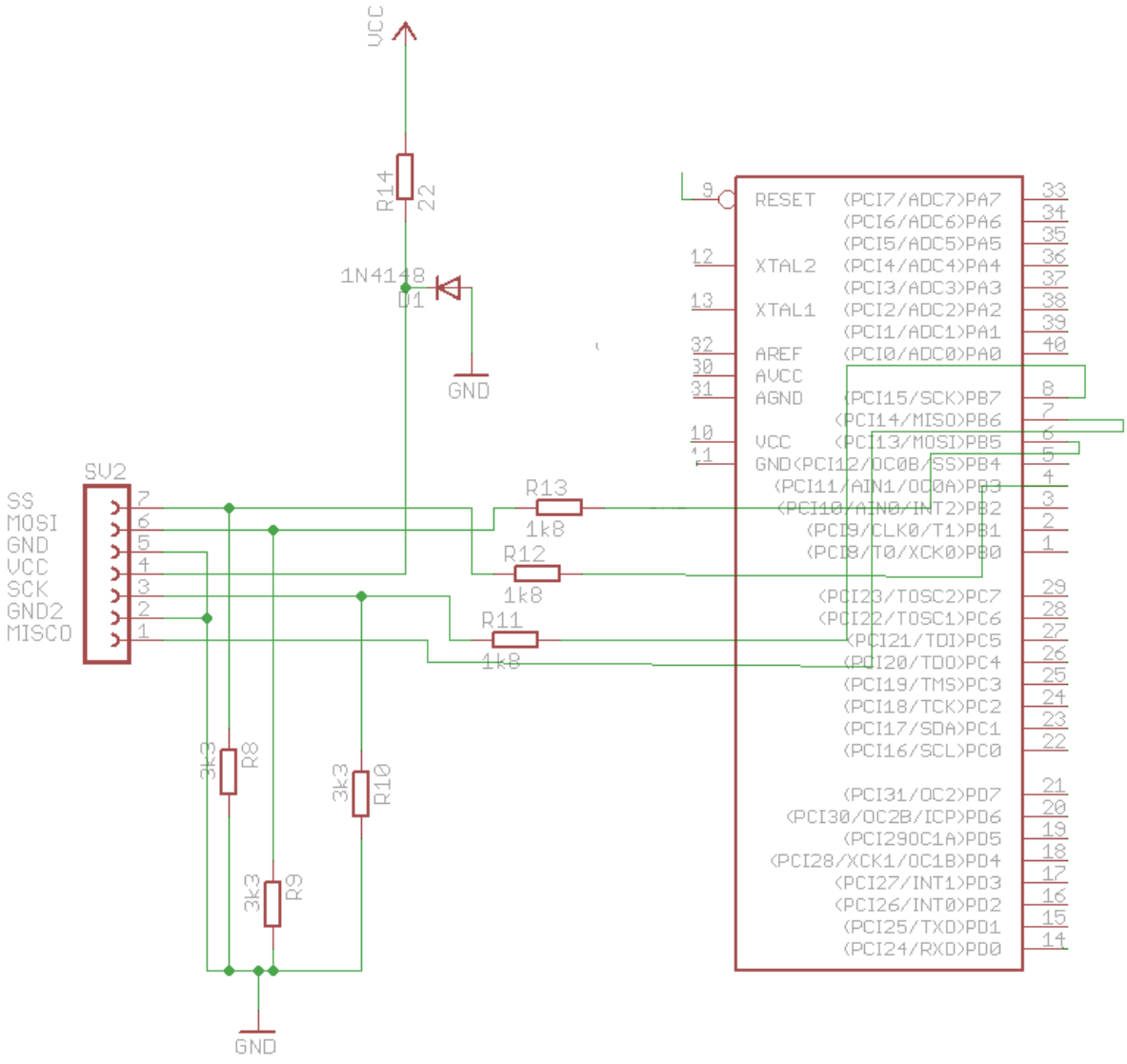


Abbildung 2-2-21: Die SD-Karte

Der VGA-Anschluss:

Um eine korrekte VGA-Ausgabe zu erzeugen, werden ein paar zusätzliche Bauteile benötigt. Zunächst wurden PIN D4 und D6 als H- und V-Sync-Signal konfiguriert und mit den VGA-Pins 14 und 13 verbunden.

Um das Umschalten zwischen Schrift- und Hintergrundfarbe zu ermöglichen, wird die Farbe in einem 8-Bit-Feld auf dem Microcontroller gespeichert und über PORT C ausgegeben. Über Port A werden die Zeichen an das Schieberegister ausgegeben und durch die serielle Weiterleitung der Bits erfolgt dann die Auswahl der entsprechenden Farbe.

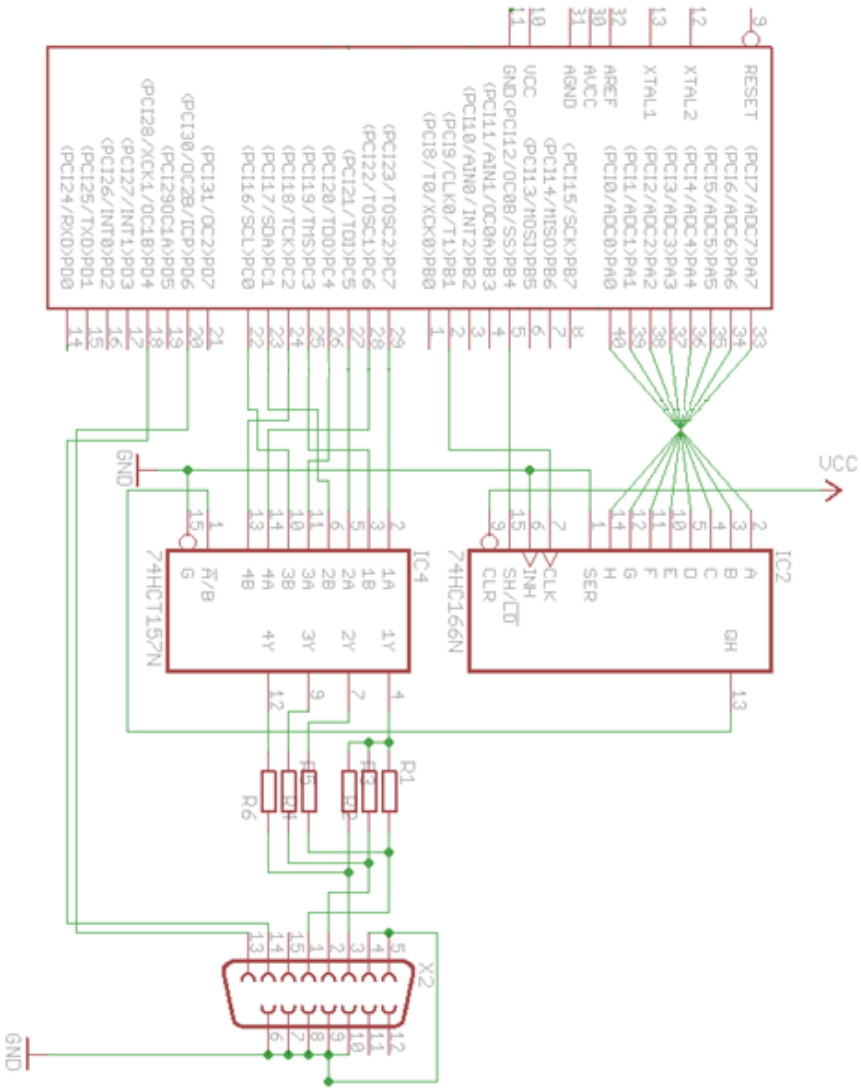


Abbildung 2-2-22: Die VGA-Ausgabe

Die gesamte Schaltung:

Zum Schluss wird nur noch eine Programmierschnittstelle benötigt. Nachdem diese eingefügt wurde, ergibt sich das folgende Schaltbild für die komplette Platine:

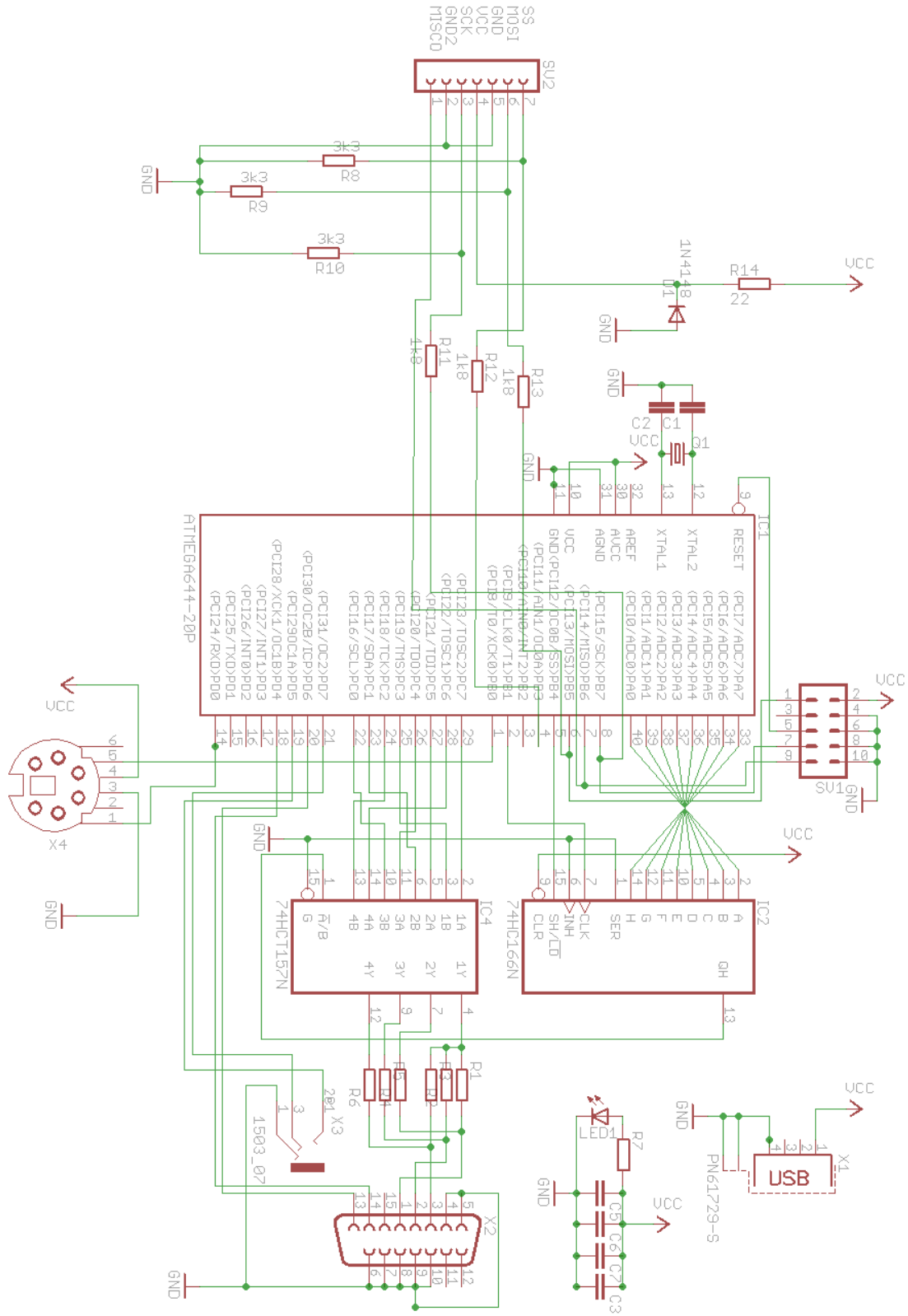


Abbildung 2-2-23: Die gesamte Schaltung

3 Die Software

Da die meisten Bestandteile bereits in früheren Projekten unabhängig voneinander entwickelt wurden, wird im Folgenden lediglich ein grober Überblick über die verschiedenen Arbeiten gegeben, dabei werden kurz die vorgenommenen Änderungen aufgeführt.

3.1 Avid

Bei diesem Teil des Programms war für mich nicht mehr viel zu tun, da es bereits Funktionen zur farbigen Ausgabe von Zeichen sowie der Generierung eines Audiosignals beinhaltet.
Um die Bibliothek zu verwenden, musste lediglich der Sourcecode dem Projekt hinzugefügt werden. Dadurch stehen „avid.c“ bzw. „avid.h“ mit den folgenden Funktionen zur Verfügung:

- avid_init : muss am Anfang des Programms aufgerufen werden, um die Videoausgabe zu initialisieren
- avid_setfont: setzt die Zeichentabelle, die zur Ausgabe des Videosignals verwendet wird
- avid_getfont: gibt den Zeiger zurück, der auf die aktuelle Zeichentabelle zeigt
- avid_getwidth: gibt die Breite des Bildschirms zurück
- avid_getheight: gibt die Höhe des Bildschirms zurück
- avid_setbgcolor: setzt die Hintergrundfarbe
- avid_setcolor: setzt die Textfarbe
- avid_setbgcolroat: setzt die Hintergrundfarbe an einer bestimmten Stelle des Bildschirms auf den übergebenen Wert
- avid_setcolorat: setzt die Textfarbe an einer bestimmten Stelle auf den übergebenen Wert
- avid_set : kopiert das übergebene Zeichen an die angegebene Stelle im Videoram
- avid_setbuff: kopiert eine übergebene Zeichenkette an einen bestimmten Bereich im Videoram
- avid_get: gibt das Zeichen an der übergebenen Stelle im Videoram zurück
- avid_getptr : gibt einen Zeiger auf die angegebenen Stelle im Videoram zurück
- avid_getbgcolor : gibt die aktuelle Hintergrundfarbe zurück

- avid_getcolor : gibt die aktuelle Textfarbe zurück
- avid_getcolorcomplete : gibt die komplette Farbe zurück
- avid_getbgcolorat : gibt die Hintergrundfarbe an einer bestimmten Stelle zurück
- avid_getcolorat : gibt die Textfarbe an einer bestimmten Stelle zurück
- avid_cls: setzt den kompletten Videoram auf das übergebene Zeichen
- avid_waitsync: wartet auf das nächste V-Sync-Signal
- avid_setframecounter: setzt den aktuellen Seitenzähler
- avid_getframecounter: gibt den aktuellen Seitenzähler zurück
- avid_setbeep: setzt den Sound Prescaler auf den übergebenen Wert (0-0xFFFF), wodurch der aktuell erzeugte Ton verändert wird
- avid_getbeep: gibt den aktuellen Sound Prescaler zurück

3.2 Femto

„Femto“ bezeichnet die Programmiersprache, die von Simeon Maxime [2] im Zuge einer Projektarbeit entwickelt wurde.
Dieses Projekt beinhaltet bereits die nötigen Softwarebausteine zur Realisierung einer Programmierumgebung, den Anschluss einer Tastatur über die PS/2-Schnittstelle sowie eine Schwarz-Weiß-Ausgabe über den VGA-Anschluss unter der Verwendung einer älteren Version der Avid-Bibliothek.

3.2.1 Anpassungen an der Videoausgabe des Interpreters

Die nächsten Schritte bestanden daraus, die alte Avid-Version durch eine aktuelle zu ersetzen, um so den Grundstein für die Farbausgabe zu legen.
Anschließend mussten alle Funktionen der Programmierumgebung, die auf die Videoschnittstelle zugreifen, für die erweiterte Farbausgabe angepasst werden.
Hier bestand das größte Problem darin, dass Zeichen- und Zahlenketten, die auf dem Bildschirm ausgegeben werden sollten, bisher einfach hintereinander in den Videospeicher geschrieben werden konnten, da die Farbkomponente fehlte.

Funktionen zur Videoausgabe

In der neuen Avid-Bibliothek ist der Speicher so aufgebaut, dass hinter jedem Zeichen die Hintergrund- und Textfarbe eingefügt wird.

Hierfür muss jede Zeichenkette in ihre einzelnen Zeichen unterteilt und diese in einer Schleife nacheinander mit der dazugehörigen Farbe in den Videoram kopiert werden.

```
inline void screenPrintStr(char *str, uint16_t length) {
    if((screenCursor+length>=screenHeight*screenWidth)||(*str=='\n'&&(screenCursor+screenWidth>=screenHeight*screenWidth))) {
            waitForEnter();
            screenClear();
            }
            while(length>0) {
                if(*str == '\n') {
                screenCursor += screenWidth-(screenCursor%screenWidth);
            } else {
                *(avid_videoram+(2*screenCursor)) = *str;
                *(avid_videoram+(2*screenCursor)+1) = avid_getcolor_complete();
                screenCursor++;
            }

            str++;
            length--;
        }
}
```

Abbildung 3-1: Beispielcode zur Ausgabe einer Zeichenkette

Der auszugebende Wert wird in die einzelnen Zeichen mit der Größe von einem Byte unterteilt und zusammen mit dem Farbwert in den Videospeicher geschrieben. Dies passiert solange, bis die komplette Kette übergeben ist.

Die Funktionen „screenPrintInt“, „screenPrintFloat“ und „priv_makeScreenSpace“ funktionieren nach dem gleichen Prinzip und wurden entsprechend angepasst.

Getter-Funktionen des Interpreters

Zu den Ausgabefunktionen kommen die Getter-Funktionen hinzu, die auf den Videospeicher zugreifen.Diese mussten ebenfalls, wie in Abbildung 3-3 zu sehen, an den veränderten Videospeicher angepasst werden.

```
66  inline uint8_t screenGet(uint16_t pos) {
67      if(pos>=screenWidth*screenHeight) {
68          return 0;
69      } else {
70          return *(avid_videoram+2*pos);
71      }
72  }
73
```

Abbildung 3-2: Angepasste Abfrage an neue Viedeospeichergröße

Verhindern des Überlaufens des Videospeichers

Abgesehen von der veränderten Größe des Videospeichers trat ein weiteres Problem mit der bisherigen Ausgabe aufgefallen:

Wenn der Inhalt von Dateien direkt auf dem Bildschirm ausgegeben wurde und ihr Inhalt wesentlich größer war, als das, was auf einer Seite angezeigt werden kann, ist regelmäßig der Speicherplatz ausgegangen.

Daher habe ich die Ausgabefunktionen so angepasst, dass das Programm angehalten wird, sobald absehbar ist, dass die nächste Eingabe außerhalb des sichtbaren Bereichs liegen wird.

In diesem Fall kann man entweder durch drücken von „Enter“ den Lesevorgang wieder anstoßen oder mit „Escape“ die Ausgabe beenden und in die aufrufende Funktion zurückkehren.

Wählt man die erste Option, wird die „print“-Funktion weiter ausgeführt bis die Ausgabe beendet ist oder erneut die Grenzen des Videospeichers erreicht sind.

```
inline int waitForEnter(){
    int c;
    while(1){
            if((c=PS2_getchar()) != -1) {
                switch(c) {
                    case KEYCODE_ENTER:
                        return 0;
                        break
                    case KEYCODE_ESC:
                        return 1;
                        break;
                }
            }
    }
}
```

Abbildung 3-3: Auszug aus screen.h: waitForEnter()

Trennen der Videoausgabe und des Buffers

Die letzten Änderungen an der Videoausgabe des Interpreters dienen dazu, den Buffer, in dem alle eingegebenen Zeichen und der Programmcode gespeichert werden, von den Ausgaben auf dem Bildschirm zu trennen.

So sollen alle Tastatureingaben und Programmcodes sowohl in den Buffer geschrieben als auch auf dem Bildschirm angezeigt werden.

Im Gegensatz dazu darf aber z. B. eine Statusmeldung oder Ähnliches auf dem Bildschirm erscheinen, würde jedoch im Buffer nur stören und darf somit nicht dorthin geschrieben werden.

Die beiden Speicher waren bisher auch schon voneinander getrennt, allerdings musste „refreshScreen“ aufgerufen werden, um über die Tastatur eingegebene Zeichen sehen zu können, Dadurch wurde nur der Inhalt des Buffers ausgegeben, womit Daten aus dem Videospeicher nicht mehr zu sehen waren.

Daher wird nun bei jedem Zeichen, das in den Buffer geschrieben wird, gleichzeitig die entsprechende Funktion zur Bildschirmausgabe aufgerufen, statt nach jedem Schleifendurchlauf in dem Hauptprogramm den kompletten Speicherinhalt des Interpreters neu zu laden.

```
void insertCharacter(char c) {
    if(getBufferFreeSize() > 0) {
        memmove(cursor+1, cursor, charsAfter(cursor));
        charsInBuffer++;
        *cursor++ = c;
        screenPrintStr(&c,1);
    }
    cursorVirtualX=getPositionInLine(cursor);
}
```

Abbildung 3-4: Auszug aus buffer.c

Entfernen von Zeichen aus dem Videospeicher

Eine weitere neue Funktion, ist das Löschen von Zeichen direkt aus dem Videospeicher. Bisher war es nur über die Tasten „Delete“ und „Backspace“ möglich, Daten aus dem Buffer zu löschen und durch ein erneutes Laden diese auch vom Bildschirm zu entfernen.

Da die Shell der SD-Karte jedoch unabhängig vom Buffer arbeiten muss, habe ich dementsprechend eine solche Funktion implementiert.

Hier wird immer das letzte Zeichen des Videospeichers entfernt und damit auch die Änderung unmittelbar auf dem Bildschirm ausgegeben.

```
inline void screenDelete(){
    if(screenCursor>0){
    screenCursor--;
    *(avid_videoram+(2*(screenCursor)))=0x00;
    }
}
```

Abbildung 3-5: screenDelete aus screen.h

Einbetten der Funktionen in Femto

Abschließend waren noch einigeAnpassungen notwendig, um die neuen Funktionen in der Programmiersprache Femto auch verwenden zu können.

Damit der Interpreter die neuen Schlüsselwörter erkennt, können mit einem in der dazugehörigen Projektarbeit [2] enthaltenen Generator die nötigen Hashwerte generiert werden.
Nun müssen nur noch die neuen Funktionen in „builtinFunctions.c“ und „builtinFunctions.h“ implementiert werden, welche die notwendigen Befehle der Avid-Bibliothek ausführen.

```
void builtinColor(uint8_t *numParams){
    unsigned char color = popIntParam(numParams);
    screenSetColor(color);
}

void builtinBgcol(uint8_t *numParams){
    unsigned char color = popIntParam(numParams);
    screenSetBgColor(color);
}


void builtinColat(uint8_t *numParams){
    unsigned char col = popIntParam(numParams);
    unsigned int y = popIntParam(numParams);
    unsigned int x = popIntParam(numParams);


    screenSetColorAt(x,y,col);
}

void builtinBgcolat(uint8_t *numParams){
    unsigned char col = popIntParam(numParams);
    unsigned int y = popIntParam(numParams);
    unsigned int x = popIntParam(numParams);


    screenSetBgColorAt(x,y,col);
}

void builtinTone(uint8_t *numParams){
    unsigned char channel = popIntParam(numParams);
    unsigned int tone = popIntParam(numParams);

    avid_setbeep(tone, channel);
}
```

Abbildung 3-6: neue "builtinFunctions"

Durch diese Änderungen kann der Interpreter mit den zusätzlichen Funktionen verwendet werden, wie in der Projektarbeit von Simeon Maxime beschrieben.

3.3 ELM-Chans FATF und SD-Karte

Die letzte und aufwendigste Erweiterung war das Integrieren einer SD-Karte in die vorhandenen Programme.

Da es zu diesem Thema schon diverse Vorarbeiten gibt, werden in diesem Projekt, die c-Bibliothek von ElmChan zur Verwaltung von FAT-Dateisystemen zu verwenden.[3]
Es wird zwar nur der Teil für SD-Karten verwendet, theoretisch ist die Bibliothek jedoch auch mit MMC und USB kompatibel.
Das Modul nur die Middleware bietet, somit müssen zusätzlich noch Low-Level-I/O-Funktionen und eine Anwendung zur Verwertung der erhaltenen Daten programmiert werden.

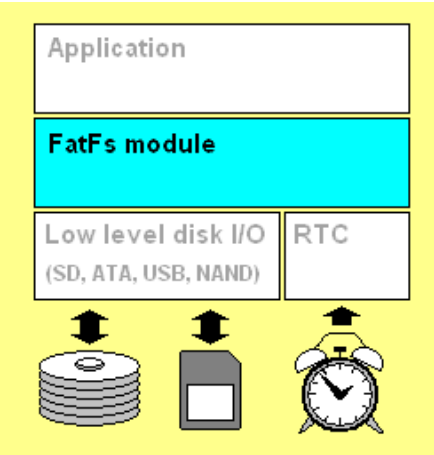


Abbildung 3-7: Einbindung der FATF-Module [3]

3.3.1 I/O-Funktionen

Im I/O-Interface der Bibliothek werden verschiedene Funktionen verwendet, welche zunächst noch implementiert werden mussten.
Alle diese Funktionen haben einen Eingabeparameter vom Typ Byte, mit dem die Art des zu verwaltenden Speichermediums ausgewählt wird. Da in meinem Projekt lediglich eine SD-Karte angeschlossen wird, darf hier immer nur eine 0 übergeben werden.
Um die Bibliothek also verwenden zu können, mussten folgende Funktionen implementiert werden:
disk_write:
Mit dieser Funktion kann eine bestimmte Anzahl von Datenblöcken auf die SD-Karte geschrieben werden. Hierzu wird zuerst überprüft, ob die Initialisierung bereits beendet ist. Anschließend werden entsprechend der Größe der zu schreibenden Datenmenge das nötige Kommando und danach die Datenblöcke an die SD-Karte gesendet.

Als Rückgabewert erhält man ein Statusflag, das mitteilt, ob die Übertragung erfolgreich war.

```
DRESULT disk_write (
    BYTE drv,           /* Physical drive nmuber (0) */
    const BYTE *buff,   /* Zeiger auf die zu schreibenden Daten */
    DWORD sector,       /* Anfangssektor */
    UINT count          /* Anzahl der Sektoren*/
)
{
        if (disk_status(drv) & STA_NOINIT){
        screenPrintStr("\ncard not ready",15);
         return RES_NOTRDY;
         }
    if (!(CardType & CT_BLOCK)) sector *= 512;  /* Anfangssektor ggf. in Byteadresse umwandeln */

    if (count == 1) {   /* einzelnen Block schreiben */
        if ((send_cmd(CMD24, sector) == 0)
            && xmit_datablock(buff, 0xFE)){
            count = 0;
        }
    }
    else {              /* Multiple block write */
    screenPrintStr("\nwriting multiple blocks",24);
        if (CardType & CT_SDC) send_cmd(ACMD23, count);

        if (send_cmd(CMD25, sector) == 0) { /* mehrere Blöcke schreiben */
            do {
                if (!xmit_datablock(buff, 0xFC)) break;
                buff += 512;
            } while (--count);
            if (!xmit_datablock(0, 0xFD))
                count = 1;
        }
    }
    deselect();

    return count ? RES_ERROR : RES_OK;
}
```

Abbildung 3-8: Funktion disk_write

disk_initalize:

Mit „dist_initialize“ wird die in Kapitel 2.4.4 beschriebene Initialisierung durchgeführt. Währenddessen wird auf dem Bildschirm der aktuelle Status ausgegeben.

```
DSTATUS disk_initialize (
    BYTE drv        /* Physical drive nmuber (0) */
)
{
    BYTE n, ty, cmd, buf[4];
    UINT tmr;
    DSTATUS s;

    screenPrintStr("\ninitialize card",17);
    if (drv) return RES_NOTRDY;

    _delay_ms(10);          /* 10ms */
    CS_INIT(); CS_H();      /* Initilaisiere CS*/
    CK_INIT(); CK_L();      /* Initialisiere Clock*/
    DI_INIT();              /* Initialisiere DI*/
    DO_INIT();              /* Initialisiere DO */

    /* SPI initialisieren
    Während der Initialisierung der SD-Karte darf die
    Clock Frequenz nicht über 400 MHZ sein*/
    SPCR = (0 << SPIE) | /* SPI Interrupt Enable */
           (1 << SPE)  | /* SPI Enable */
           (0 << DORD) | /* Data Order: MSB first */
           (1 << MSTR) | /* Master mode */
           (0 << CPOL) | /* Clock Polarity: SCK low when idle */
           (0 << CPHA) | /* Clock Phase: Bei steigender Tackflanke*/
           (1 << SPR1) | /* Clock Frequency: f_OSC / 128 */
           (1 << SPR0);
    SPSR &= ~(0 << SPI2X); /* Keine doppelte Clock Frequenz */

    for (n = 10; n; n--) rcvr_mmc();    /* 80 dummy Clocks senden bis die Karte bereit ist Daten zu empfangen*/

    ty = 0;
    if (send_cmd(CMD0, 0) == 1) {           /* Enter Idle state */
        if (send_cmd(CMD8, 0x1AA) == 1) {   /* SDv2? */
            screenPrintStr("\nSDv2",5);

            for(BYTE i=0;i<4 ; i++)buf[i]=rcvr_mmc();                           /* R7 Antwort der Karte Speichern*/

            if (buf[2] == 0x01 && buf[3] == 0xAA) {     /* Überprüfen ob die Karten im Bereich von 2.7-3.6V arbietet*/
                for (tmr = 1000; tmr; tmr--) {          /* Warten bis IdleState wieder verlassen wurde*/
                    if (send_cmd(ACMD41, 1UL << 30) == 0) break;

                    _delay_ms(1);
                }
                if (tmr && send_cmd(CMD58, 0) == 0) {   /* Überprüfen des CCS bit */
                    for(BYTE i=0; i<4; i++)buf[i]=rcvr_mmc();
                    ty = (buf[0] & 0x40) ? CT_SD2 | CT_BLOCK : CT_SD2;  /* SDv2 */
                    screenPrintStr("\ncard is valid",14);
                }
            }
        } else {                            /* SDv1 or MMCv3 */
            screenPrintStr("\nSDv1",5);
            if (send_cmd(ACMD41, 0) <= 1)   {
                ty = CT_SD1; cmd = ACMD41;  /* SDv1 */
            } else {
                ty = CT_MMC; cmd = CMD1;    /* MMCv3 */
            }
            for (tmr = 1000; tmr; tmr--) {          /* Warten bis IdleState wieder verlassen wurde*/
                if (send_cmd(cmd, 0) == 0) break;
                _delay_ms(1);
            }
            if (!tmr || send_cmd(CMD16, 512) != 0)  /* Zur Sicherheit Blockgröße für den R/W-Prozess auf 512 Byte setzen*/
                ty = 0;
        }
    }
    CardType = ty;
    s = ty ? 0 : STA_NOINIT;
    Stat = s;

    deselect();             //Nach der Initialisierung kann die Clock mit maximal möglicher
                            //Geschwindigkeit betrieben werden
    SPCR = (0 << SPIE) | /* SPI Interrupt Enable */
           (1 << SPE)  | /* SPI Enable */
           (0 << DORD) | /* Data Order: MSB first */
           (1 << MSTR) | /* Master mode */
           (0 << CPOL) | /* Clock Polarity: SCK low when idle */
           (0 << CPHA) | /* Clock Phase: sample on rising SCK edge */
           (0 << SPR1) | /* Clock Frequency: f_OSC / 2 */
           (0 << SPR0);
    SPSR &= ~(1 << SPI2X);
    return s;
}
```

Abbildung 3-9: Funktion disk_initialize

disk_status:

Diese Funktion gibt den aktuellen Status des Mediums zurück. Vorgesehen ist, dass eine Kombination von bis zu drei Flags resultiert.

- STA_NOINIT: 1, wenn die SD-Karte nicht initialisiert wurde
- STA_NODISK: 1, falls kein Medium im Laufwerk ist
- STA_NOPROTECT: 1, falls das Medium schreibgeschützt ist

„STA_NODISK“ wird nie gesetzt, da bei einem nicht vorhandenen Medium bereits die Initialisierung fehlschlägt.

„STA_NOPROTECT“ wird nicht benötigt, da es keine zusätzliche Leitung gibt, um den Status des Schreibschutzes zu überprüfen. Dieser wird bei manchen SD-Karten über einen zusätzlichen Pin gesetzt. Somit fällt diese Funktion relativ einfach aus. Es wird lediglich geprüft, ob die Karte initialisiert wurde oder nicht und der entsprechende Status zurückgegeben.

```
368  DSTATUS disk_status (
369      BYTE drv
370  )
371  {
372      if (drv) return STA_NOINIT;
373
374      return Stat;
375  }
376
```

Abbildung 3-10: Code zu disk_status

disk_read:

Die Funktion wird benötigt, um Daten von der SD-Karte zu lesen. Wie in Kapitel 2.4 beschrieben wird entsprechend der Größe des zu lesenden Datensatzes entweder das Kommando CMD17 oder das Kommando CMD18 gesendet und anschließend in einer while-Schleife die angegebene Anzahl an Datenblöcken empfangen. Abschließend wird im Falle einer Übertragung von mehreren Datenblöcken das Kommando CMD12 gesendet, um den Datentransfehr zu unterbrechen.

```
DRESULT disk_read (
    BYTE drv,           /* Physical drive nmuber (0) */
    BYTE *buff,         /* Buffer in dem die Daten gespeichert werden*/
    DWORD sector,       /* Anfangssektor */
    UINT count          /* Anzahl der zu lesenden Sektoren*/
)
{
    BYTE cmd;

    if (disk_status(drv) & STA_NOINIT) return RES_NOTRDY;
    if (!(CardType & CT_BLOCK)) sector *= 512;  /* ggf. Anfangsektor in Byteadresse umwandeln */

    cmd = count > 1 ? CMD18 : CMD17;            /*  READ_MULTIPLE_BLOCK : READ_SINGLE_BLOCK */
    if (send_cmd(cmd, sector) == 0) {
        do {
            if (!rcvr_datablock(buff, 512)){
                screenPrintStr("\nreading datablock failed",25);
                 break;
                 }
            buff += 512;
        } while (--count);
        if (cmd == CMD18) send_cmd(CMD12, 0);   /* Übertragung stoppen*/
    }
    deselect();

    return count ? RES_ERROR : RES_OK;
}
```

Abbildung 3-11: Funktion disk_read

get_fattime:

Normalerweise gibt die Funktion das aktuelle Datum mit Uhrzeit zurück um z. B. beim Anlegen einer Datei den Zeitpunkt der Erstellung festzuhalten. Da es jedoch nicht möglich ist, über den Microcontroller das aktuelle Datum zu ermitteln, wird lediglich ein Standardwert zurückgegeben.

```
4594    DWORD get_fattime (void)
4595    {
4596        /* Returns current time packed into a DWORD variable */
4597        return    ((DWORD)(2013 - 1980) << 25)  /* Year 2013 */
4598                | ((DWORD)7 << 21)              /* Month 7 */
4599                | ((DWORD)28 << 16)             /* Mday 28 */
4600                | ((DWORD)0 << 11)              /* Hour 0 */
4601                | ((DWORD)0 << 5)               /* Min 0 */
4602                | ((DWORD)0 >> 1);              /* Sec 0 */
4603    }
```

Abbildung 3-12: Funktion get_fattime

disk_ioctl:

Die Funktion soll die Möglichkeit bieten, weitere Statusabfragen und administrative Aktionen durchzuführen. Dazu werden ein Kommando und ggf. die dazugehörigen Parameter übergeben, um die entsprechende Operation auszuwählen.

Welche Befehle die Funktion verwalten kann, ist in der Dokumentation zur FatF-Bibliothek aufgeführt.

Die folgenden Möglichkeiten können gewählt werden:

- CTRL_SYNC: Diese Funktion überprüft, ob die Karte den letzten Schreibprozess beendet hat
- GET_SECTOR_COUNT: Diese Funktion gibt die Anzahl der auf der Karte verfügbaren Sektoren zurück
- GET_BLOCK_SIZE: Diese Funktion gibt die Blockgröße der Karte als Anzahl an Sektoren zurück.

```
DRESULT disk_ioctl (
    BYTE drv,       /* Physical drive nmuber (0) */
    BYTE ctrl,      /* Control code */
    void *buff      /* Buffer to send/receive control data */
)
{
    DRESULT res;
    BYTE n, csd[16];
    DWORD cs;


    if (disk_status(drv) & STA_NOINIT) return RES_NOTRDY;   /* Überprüfen ob Karte vorhanden */

    res = RES_ERROR;
    switch (ctrl) {
        case CTRL_SYNC :        /* Sicherstellen, dass kein anderer Prozess au fdie Karte zugreift */
            if (select()) res = RES_OK;
            break;

        case GET_SECTOR_COUNT : /* anzahl der Sektoren ermitteln*/
            if ((send_cmd(CMD9, 0) == 0) && rcvr_datablock(csd, 16)) {
                if ((csd[0] >> 6) == 1) {   /* SDC ver 2.00 */
                    cs = csd[9] + ((WORD)csd[8] << 8) + ((DWORD)(csd[7] & 63) << 16) + 1;
                    *(DWORD*)buff = cs << 10;
                } else {                    /* SDC ver 1.XX or MMC */
                    n = (csd[5] & 15) + ((csd[10] & 128) >> 7) + ((csd[9] & 3) << 1) + 2;
                    cs = (csd[8] >> 6) + ((WORD)csd[7] << 2) + ((WORD)(csd[6] & 3) << 10) + 1;
                    *(DWORD*)buff = cs << (n - 9);
                }
                res = RES_OK;
            }
            break;

        case GET_BLOCK_SIZE :
            *(DWORD*)buff = 128;
            res = RES_OK;
            break;

        default:
            res = RES_PARERR;
    }

    deselect();

    return res;
}
```

Abbildung 3-13: Funktion disk_ioctl

3.3.2 Die SD-Shell

Um den Inhalt der SD-Karte auch über den Microcontroller steuern zu können, wird noch eine Oberfläche benötigt.

Da als Eingabegerät nur die Tastatur zur Verfügung steht, habe ich mich hier für die Implementierung einer vereinfachten Shell-Umgebung entschieden.

Diese kann aus dem Hauptprogramm über „F2“ aufgerufen und durch Drücken von „Escape“ wieder verlassen werden.

Bei Betreten des Unterprogramms wird zunächst die SD-Karte initialisiert und anschließend erfolgt die dauerhafte Abfrage der Tastatur in einer while-Schleife. Während der Ausführung

dieses Programmteils ist die Ausgabe der gedrückten Tasten deaktiviert und nur diejenigen lösen eine Aktion aus, denen eine Funktion zugeordnet wurde.
Wird eine dieser Funktionen aktiviert, muss zunächst noch ein Parameter eingegeben und mit „Enter“ bestätigt werden. Dabei gilt, dass Pfade immer relativ angegeben werden.
Alle implementierten Optionen sind in der folgenden Tabelle kurz zusammengefasst und werden anschließend kurz erläutern.

Taste	Parameter	Funktion	Beschreibung
w	Dateiname	sd_puts_file	Schreibt den Inhalt des Buffers in die angegebene Datei auf der SD-Karte
r	Dateiname	sd_reads_file	Schreibt den Inhalt der angegebenen Datei in den Buffer
s	Keine	sd_scan_files	Gibt den Inhalt des aktuellen Ordners aus
c	Ordnername	sd_create_dir	Erstellt einen Ordner mit dem angegebenen Namen
m	Ordnerpfad	sd_move	Navigiert zu dem angegebenen Pfad
p	Dateiname	sd_print_file	Gibt den Inhalt der Datei auf dem Bildschirm aus
e	Dateiname	sd_write_eeprom	Speichert den Dateiinhalt im EEPROM
d	Datei-/Ordnername	sd_delete	Löscht die Datei/den Ordner
t	Dateiname	sd_truncate	Löscht den Inhalt der Datei
u	Datei-/Ordnername	sd_rename	Benennt die Datei/den Ordner um oder verschiebt sie/ihn

Tabelle 6: Beschreibung der Shell Funktionen

Allgemeines

Zunächst folgen ein paar allgemeine Informationen zu den Funktionen und ihren Eigenschaften.
Über die Funktion „sd_move“ kann innerhalb des Dateisystems der SD-Karte navigiert werden. Um die Eingabe intuitiver und möglichst nah an bekannten Shell-Umgebungen zu

halten, wird hierfür eine relative Pfandangabe verwendet. Somit werden alle Eingaben relativ zum aktuell gewählten Ordner interpretiert.
Damit nun auf Dateien und Ordner zugegriffen werden kann, müssen sie mit der Funktion „f_open“ bzw. „f_opendir“ ausgewählt werden. Hier werden immer ein Datei-/Pfadname und ein Statusflag, in dem festgelegt wird, ob das Objekt für den lesenden oder schreibenden Zugriff geöffnet werden soll, übergeben. Als Rückgabeparameter dient ein Zeiger auf ein „File“ oder „Directory“ .
Dies sind in ElmChans Bibliothek [3] festgelegt Objekte, die notwendige Informationen wie Datei-/Ordnername, Speicherstelle auf der SD-Karte oder Ähnliches beinhalten.
Nachdem das Objekt geöffnet wurde, kann die entsprechende Operation ausgeführt werden. Dies geschieht jedoch zunächst nur im Speicher des Microcontrollers. Erst mit Aufruf der Funktion „f_sync“ werden die Änderungen auf die Karte übertragen.
Im Allgemeinen wird mit Abschluss einer solchen Aktion allerdings die Funktion „f_close“ aufgerufen, welche die Synchronisation implizit durchführt und die Datei/den Ordner wieder freigegibt.
Als Letztes wird in dem Fall, dass eine der Funktionen nicht korrekt ausgeführt werden kann, die Funktion „sd_error_msg“ aufgerufen. Sie ordnet den verschiedenen (von der FatF-Bibliothek [3] generierten) Statusbits konkreten Statusmeldungen zu und gibt sie auf dem Bildschirm aus.

Schreiben in eine Datei

Um in eine Datei zu schreiben, muss diese mit dem Statusflag FA_WRITE geöffnet werden. Damit sichergestellt ist, dass das ausgewählte Objekt auch vorhanden ist, wird zusätzlich noch die Funktion (?) FA_CREATE_ALWAYS übergeben, wodurch die Datei erstellt wird, falls sie noch nicht existiert.
Anschließend wird der gesamte Inhalt des Buffers in das Fileobjekt geschrieben und zum Abschluss durch „f_close“ mit der Karte synchronisiert.

```
void sd_puts_file(char* filename)
{


    screenPrintStr("\nwriting to file ",17);
    screenPrintStr(filename, strlen(filename));
    res = f_open(&Fil, filename, FA_WRITE | FA_CREATE_ALWAYS) ;/* Open or create a file */

    if (res == FR_OK)
    {
        res=f_puts((char*)&(buffer->bufferContent),&Fil);          //write String to file
        if(res==FR_OK)
        {
            screenPrintStr("\nready",6);
        }
        else
        {
            sd_error_msg(res);
        }
        f_close(&Fil);                                  /* Close the file */
    }
    else
    {
        sd_error_msg(res);
    }
}
```

Abbildung 3-14: Funktion sd_puts_file

Lesende Funktionen

Die folgenden Funktionen greifen alle in gleicher Weise auf eine Datei in der SD-Karte zu. Sie unterscheiden sich lediglich durch der Form, in der die erhaltenen Informationen ausgegeben werden.

Zunächst wird die angeforderte Datei mit dem Statusflag FA_READ geöffnet und in einer Schleife zeilenweise über den Inhalt iteriert.

In jedem Schleifendurchlauf wird dann die entsprechende Ausgabefunktion durchgeführt.

- Schreiben in den Buffer

```
void sd_reads_file(char* filename)
{

    char line[64]; /* Line buffer */
    uint16_t lineSize =0;

    screenPrintStr("\nread File: ",12);
    screenPrintStr(filename,strlen(filename));

    clear_buffer();

    res= f_open(&Fil, filename, FA_READ);

    if( res==FR_OK)
    {

        while (f_gets(line, sizeof line, &Fil)) /* read data from file */
        {
            lineSize=strlen(line);
            moveSdToBuffer(line,lineSize);

        }
        f_close(&Fil);
        refreshScreen((char*)avid_videoram, MAXVGAWIDTH, MAXVGAHEIGHT);
    }
    else
    {
         sd_error_msg(res);
     }
}
```

Abbildung 3-15: Funktion „sd_reads_file“

Inhalt einer Datei auf dem Bildschirm ausgeben

Die Funktion „sd_print_file“ gibt den Inhalt der übergebenen Datei als Text auf dem Bildschirm aus.

```
void sd_print_file(char* filename)
{
    char line[64]; /* Line buffer */


    screenPrintStr("\nread File: ",12);
    screenPrintStr(filename,strlen(filename));

    res= f_open(&Fil, filename, FA_READ);
    if( res==FR_OK)
    {
        while (f_gets(line, sizeof line, &Fil))/* read data from file */
        {
            if(screenPrintStr("\n",1);)return;
            if(screenPrintStr(line,strlen(line));)return;

        }
        f_close(&Fil);
    }
    else
    {
        sd_error_msg(res);
     }

}
```

Abbildung 3-16: Funktion „sd_print_file“

Dateiinhalt in den EEPROM kopieren

„sd_write_eeprom“ liest die angegebene Datei Zeilenweise aus und kopiert den Inhalt in den EEPROM.

```
void sd_write_eeprom(char *filename)
{

    char line[64]; /* Line buffer */
    uint16_t lineSize =0;
    uint16_t eepromSize =0x0000;

    screenPrintStr("\nread File: ",12);
    screenPrintStr(filename,strlen(filename));
    res= f_open(&Fil, filename, FA_READ);

    if(res==FR_OK)
    {
        screenPrintStr("\nreading File", 13);

        while (f_gets(line, sizeof line, &Fil)) /* read data from file */
        {
            lineSize=strlen(line);
            eeprom_write_block(line, (void*)eepromSize, lineSize);
            eepromSize+=lineSize;
        }
        f_close(&Fil);
    }
    else
    {
        sd_error_msg(res);
     }
}
```

3-17 Funktion sd_write_eeprom

Ausgeben des Ordnerinhalt

Als Vorlage zum Scannen des Ordners diente ein Beispielaufruf aus der Dokumentation zur FatF-Bibliothek.[3]

Die Funktion iteriert über alle im aktuellen Ordner hinterlegten Items und gibt deren Namen und die absolute Pfadangabe auf dem Bildschirm aus.

```
void sd_scan_files ()
{
    FILINFO fno;
    DIR dir;
    TCHAR buff[20];

    int i;
    char *fn;   /* This function is assuming non-Unicode cfg. */

    f_getcwd(buff,20);

#if _USE_LFN
    static char lfn[_MAX_LFN + 1];   /* Buffer to store the LFN */
    fno.lfname = lfn;
    fno.lfsize = sizeof lfn;
#endif
    screenPrintStr("\nscanning directory ",19);
    screenPrintStr(buff,strlen(buff));
    res = f_opendir(&dir, buff);                       /* Open the directory */

    if (res == FR_OK)
    {
        i = strlen(buff);
        for (;;)
        {
            res = f_readdir(&dir, &fno);                   /* Read a directory item */
            if (res != FR_OK || fno.fname[0] == 0) break;  /* Break on error or end of dir */
            if (fno.fname[0] == '.') continue;             /* Ignore dot entry */
#if _USE_LFN
            fn = *fno.lfname ? fno.lfname : fno.fname;
#else
            fn = fno.fname;
#endif
            if (fno.fattrib & AM_DIR)
            {
                screenPrintStr("\ndirectory: ",12);
            }                    /* It is a directory */
            else
            {
                screenPrintStr("\nfile: ",7);
            }                                          /* It is a file. */
                screenPrintStr(&buff[i], strlen(&buff[i]));
                screenPrintStr("/",1);
                screenPrintStr(fn, strlen(fn));

                if (res != FR_OK) break;
                buff[i] = 0;
        }

        f_closedir(&dir);
    }
    else
    {
        sd_error_msg(res);
    }
}
```

Abbildung 3-18: Funktion „sd_scan_files“

Ordner öffnen

Zunächst wird die nötige Funktion der FatF-Bibliothek[3] aufgerufen. Um den Überblick zu behalten, welcher Ordner aktuell ausgewählt ist, wird anschließend der aktuelle Pfad auf dem Bildschirm ausgegeben.

```
void sd_move(char* path){
    TCHAR buff[20];

    screenPrintStr("\nmoving to directory ",21);
    screenPrintStr(path,strlen(path));
    res=f_chdir(path);
    if(res==FR_OK)
    {
        screenClear();

        res = f_getcwd(buff,20);
        if(res == FR_OK)
        {
            screenPrintStr("\ncurrent Directory: ",21);
            screenPrintStr(buff,strlen(buff));
        }
        else
        {
            sd_error_msg(res);
        }
    }
    else
    {
        sd_error_msg(res);
    }

}
```

Abbildung 3-19: Funktion „sd_move“

Datei/Ordner umbenennen oder verschieben

Der alte und neue Name der Datei oder des Ordners werden übergeben und die entsprechende Datei umbenannt.

Beinhaltet der neue Name als Präfix eine relative Pfadangabe, wird das Objekt an den entsprechenden Ort verschoben.

```
void sd_rename(char old_name[20], char new_name[20])
{

    screenPrintStr("\nold name: ",11);
    screenPrintStr(old_name, strlen(old_name));
    screenPrintStr("\nnew name: ",11);
    screenPrintStr(new_name, strlen(new_name));

    res = f_rename(old_name, new_name);

    if(res==FR_OK)
    {
        screenPrintStr("\nFile renamed",13);
    }
    else
    {
        sd_error_msg(res);
    }
}
```

Abbildung 3-20: Funktion „sd_rename“

Weitere Funktionen

Die folgenden Funktionen sind Hüllen, um einheitliche Namen und die Trennung der FatF-Bibliothek[3] von dem eigentlichen Programm zu erreichen.

- Löschen einer Datei/eines Ordners

```
void sd_delete(char* name)
{

    res=f_unlink(name);
    if(res==FR_OK)
    {
        screenPrintStr("\nDeletion complete",18);
    }
    else
    {
        sd_error_msg(res);
    }
}
```

Abbildung 3-21: Funktion „sd_delete“

- Löschen des Inhalts einer Datei

```
void sd_truncate(char* filename)
{
    res= f_open(&Fil, filename, FA_WRITE);

    if(res==FR_OK)
    {
        screenPrintStr("\ntruncate File", 15);
        res=f_truncate(&Fil);
        if(res==FR_OK)
        {
            screenPrintStr("\nready",7); }
        else
        {
            sd_error_msg(res);
        }
    }
    else
    {
         sd_error_msg(res);
    }
    f_close(&Fil);

}
```

Abbildung 3-22: Funktion „sd_truncate“

- Erstellen eines Unterordners

```
void sd_create_dir(char* path)
{


    screenPrintStr("\ncreating directory: ", 21);
    screenPrintStr(path,strlen(path));

    res =f_mkdir(path);

    if(res==FR_OK)
    {
        screenPrintStr("\ncreated",8);
    }
    else
    {
        sd_error_msg(res);
    }
}
```

Abbildung 3-23: Funktion „sd_create_dir“

4 Fazit

Ziel der Arbeit war es, einen eigenständigen Homecomputer zu entwickeln, der über einen eigenen Speicher verfügt, wodurch nicht mehr die Notwendigkeit besteht, entweder Programme mit jedem Start neu zu schreiben oder auf einem externen System zu speichern. Dies ist nun durch die angeschlossene SD-Karte möglich. Mit der erweiterten Farbausgabe bieten sich auch weitere Möglichkeiten zu Verwendung des Interpreters.
Denkbar wäre es z. B., eine USB-Schnittstelle für ein weiteres Speichermedium oder andere externe Geräte einzubauen.
Außerdem bietet das Projekt von ElmChan[3] die Möglichkeit, SD-Karten oder USB-Speicher in einem Stream auszulesen. Somit kann für die nächsten Erweiterungen ein Mediaplayer in Betracht gezogen werden.
Abschließend lässt sich also sagen, dass zwar die Entwicklung eines Homecomputer erfolgreich war, jedoch noch diverse weitere Funktionen denkbar sind.

5 Quellenverzeichnis

[1] Seminar "Machbarkeitsstudie Einchipcomputer", 2007. URL http://userpages.uni-koblenz.de/~physik/informatik/ECC/

[2] Simeon Maxime. Eine Programmierumgebung für den ATmega-Mikrocontroller, September 2010

[3] FatFs - Generic FAT File System Module, Januar 2014. URL http://elm-chan.org/fsw/ff/00index_e.html

[4] ATmega1284p preliminary datasheet, November 2009. URL http://www.atmel.com/images/doc8059.pdf

[5] Christoph Sallie, Benedikt Joebgen. Projektpraktikum Virtueller Flipper – Microcontroller als PS2 Tastatur, April 2012

[6] Signaldaten für 15-poligen Mini-D-Sub-Stecker. URL http://www.dharkkum.de/pics/vga_yuv.gif

[7] Wikipedia. Klinkenstecker , August 2014. URL http://de.wikipedia.org/w/index.php?title=Klinkenstecker&oldid=133213701

[8] How to Use MMC/SDC, Februar 2013. URL http://elm-chan.org/docs/mmc/mmc_e.html

[9] Dr. Merten Joost. Avidlib, August 2014

6 Anhang

Anhang 1: CD mit Sourcecode, Schaltplänen und Ausarbeitung als PDF

Erklärung

„Hiermit bestätige ich, dass die vorliegende Arbeit von mir selbständig verfasst wurde und ich keine anderen als die angegebenen Hilfsmittel – insbesondere keine im Quellenverzeichnis nicht benannten Internet–Quellen – benutzt habe und die Arbeit von mir vorher nicht in einem anderen Prüfungsverfahren eingereicht wurde. Die eingereichte schriftliche Fassung entspricht der auf dem elektronischen Speichermedium (CD-Rom).“

	Ja	Nein
Mit der Einstellung der Arbeit in die Bibliothek bin ich einverstanden.	☐	☐
Der Veröffentlichung dieser Arbeit im Internet stimme ich zu.	☐	☐

……..

(Ort, Datum) (Unterschrift)