Home
AVR
- AVR-Tutorial
- AVR-GCC-Tutorial

ARM
- LPC2000
- AT91SAM7

MSP430
FPGA, CPLD & Co.
- Grundlagen zu FPGAs
- VHDL & Co.
- Xilinx ISE

DSP
Elektronik allgemein
- SMD Löten
- Operationsverstärker
- Oszilloskop

Foren
- µC & Elektronik
- Analogtechnik
- FPGA, VHDL & Co.
- DSP
- GCC
- Codesammlung
- Markt
- Platinen
- HF, Funk & Felder
- Hausbus
- PC-Programmierung
- PC Hard- & Software
- Ausbildung & Beruf
- Offtopic
- Webseite

Chat
Buchtipps
Shop
Linksammlung
Artikelübersicht
Letzte Änderungen

Dieser Artikel
- Seite
- Diskussion
- Bearbeiten
- Versionen/Autoren

Benutzer
- 188.65.176.26
- Diskussionsseite dieser IP
- Anmelden

Suche
Ausführen
Suche

Werkzeuge
- Links auf diese Seite
- Änderungen an verlinkten Seiten
- Spezialseiten

AVR FAT32

Inhaltsverzeichnis

Der Status

[Bearbeiten]

Aktuelle Version: SVN-Repository
Bekannte Probleme: Bei Big Endian Controllern müssen größere Anpassungen gemacht werden.
Problem melden, oder sonstige Anfrage: Persönliche Nachricht, oder auch hier Thread

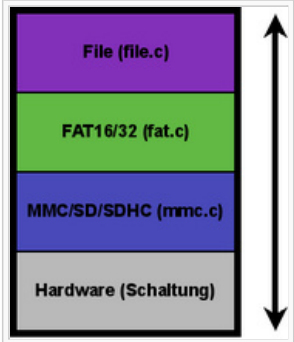
FAT 16/32 , die Bibliothek

[Bearbeiten]

Dies ist eine freie FAT 16 / 32 Bibliothek.

Die Bibliothek ist modular und besteht aus folgenden Modulen:

- File
- FAT16/32
- MMC/SD (hardwareabhängig)



Schichtmodell der Lib

Das MMC/SD-Modul ist dafür zuständig, die Kommunikation mit MMC- und SD-Karten zu managen. Wie z. B. Initialisierung der Karte oder low-level Routinen zum schreiben auf die Karte. Es ist das einzige Hardware abhängige Modul.

Das FAT16/32-Modul bietet die grundlegenden Funktionen für den FAT-Zugriff und die Initialisierung der FAT. Es dient als Middleware zwischen low-level Karten Zugriff und high-level Datei Operationen. Der größte Teil der des FAT Protokolls ist dort implementiert.

Das File-Modul bietet high-level Datei Operationen wie man sie von Dateizugriffen kennt, z. B. ffopen/ffclose um eine Datei zu öffnen oder um an eine Datei etwas anzuhängen und diese wieder zu schließen.

Es gibt die Möglichkeit über #defines in der config.h den Umfang der Lib zu bestimmen, dies wirkt sich auch extrem auf die Codegröße aus.

Die Module sind c99 C-Standard Konform .

Code einbinden

[Bearbeiten]

Die Module werden über die *.h Dateien bekannt gemacht. Zudem müssen noch die dazugehörigen *.c Dateien eingebunden werden. Es gibt mehrere Möglichkeiten dies zu tun. Einmal im Makefile im Bereich der "c source files" oder bei IDEs über einen Dialog der das Gleiche macht.

Die Funktionalität

[Bearbeiten]

In der Datei config.h gibt es Möglichkeiten die Bibliothek zu konfigurieren.
Folgende Funktionen werden unterstützt (FAT16/32):

- Lesen, Schreiben und Überschreiben von Dateien.
- Vor- und Rückspulen in Dateien.
- Anlegen von Ordnern
- Wechseln von Verzeichnissen
- Löschen von Dateien und Ordnern, bei Ordnern Rekursiv.
- Ermitteln der freien Bytes auf der Karte
- Schnelleres Schreiben und Lesen durch Multi-Block Operationen.
- Kartenunterstützung für MMC/SD/SHDC.
- Falls eine RTC vorhanden ist, kann bei den Dateioperationen die Zeitstempel Funktionalität genutzt werden.
- Anbindung der Karte über Software SPI oder Hardware SPI

Die Schalter

[Bearbeiten]

Hier ein Auszug aus der config.h mit den wichtigsten Parametern:

```
#define MMC_SMALL_FILE_SYSTEM   TRUE
#define MMC_WRITE               TRUE
#define MMC_OVER_WRITE          FALSE
#define MMC_MULTI_BLOCK         FALSE
#define MMC_SDHC_SUPPORT        TRUE
#define MMC_TIME_STAMP          FALSE

#define MMC_MAX_SPEED           TRUE

#define MMC_SOFT_SPI            FALSE

#define MMC_MAX_CLUSTERS_IN_ROW 256
```

Bedeutung der Schalter

[Bearbeiten]

MMC_SMALL_FILE_SYSTEM TRUE, oder **FALSE** bestimmt den Funktionsumfang der Lib. Es Fallen Folgende Funktionalitäten raus: ffcd(), fat_str(), ffls(), ffcdLower(), ffmkdir() und ffrm() der Teil mit Ordnern rekursiv löschen. (**FALSE** = Komplette Unterstützung)

MMC_WRITE TRUE, oder **FALSE** bestimmt ob schreib Unterstützung einkompiliert wird oder nicht. (**TRUE** = Write an)

MMC_OVER_WRITE TRUE, oder **FALSE** bestimmt ob ffwrite() mit Überschreiben Funktionalität Kompiliert wird oder nicht. Überschreiben von Dateien ist nicht so performant. Siehe auch: Interne Technik (**TRUE** = Dateien überschreiben)

MMC_MULTI_BLOCK TRUE, oder **FALSE** legt fest ob mit MultiBlock Read/Write Unterstützung Kompiliert wird oder nicht. Geht nur wenn **OVER_WRITE FALSE** ist! (**TRUE** = MultiBlock Operation)

MMC_MAX_SPEED TRUE, oder **FALSE** legt fest, ob nach der Initialisierung der MMC/SD Karte mit maximalem Speed geschrieben/gelesen wird oder nicht. Zum testen wenn man nicht sicher ist ob die Hardware die maximale Geschwindigkeit mit macht empfiehlt sich **FALSE** als Wert.

MMC_MAX_CLUSTERS_IN_ROW 1-500, gibt an, wie viele Cluster, leer oder Verkettet die zusammenhängen, gesucht werden sollen. Hier muss man den Overhead des Suchens abwägen, gegen die Zeit, die z. B. benötigt wird 500 Cluster in der Fat zu verketten. Siehe auch: Interne Technik

MMC_SDHC_SUPPORT TRUE, oder **FALSE** legt fest, ob mit SDHC Unterstützung Kompiliert wird oder ohne. Der SD Standard 2.0 wird damit unterstützt, also Karten bis 32 GB Größe.

MMC_TIME_STAMP TRUE, oder **FALSE** legt fest, ob die Zeitstempel Unterstützung mit kompiliert wird oder nicht. Wenn TRUE, dann wird das Erstelldatum und die Erstellzeit eingetragen. Bei weiteren Schreibzugriffen auf die Datei wird dann auch das Zugriffsdatum und die Zugriffszeit eingetragen. Es müssen dafür auch 2 Funktionen in der fat.c mit Code gefüllt werden, damit diese das Datum und die Zeit liefern.

MMC_SOFT_SPI TRUE, oder **FALSE** legt fest, ob mit Software SPI Unterstützung kompiliert wird, oder nicht. Wenn **TRUE**, muss in der mmc.h noch eingetragen werden welche Pins als SPI Interface benutzt werden sollen. Es werden nur Pins des selben Ports unterstützt. Also z. B. nur Pins von Port B.

Die C Funktionen

[Bearbeiten]

Das Modul FILE bietet in der Standard-Konfiguration folgende für den Nutzer interessante Funktionen:

```
unsigned char ffread(void)
void          ffwrite(unsigned char c)
void          ffwrites(unsigned char *s )
unsigned char ffopen(unsigned char name[])
unsigned char ffclose(void)
void          ffseek(unsigned long int offset)
unsigned char ffcd(unsigned char name[])
```

```
void          ffls(fptr uputs_ptr)
unsigned char ffcdLower(void)
unsigned char ffrm(unsigned char name[])
unsigned char ffmkdir(unsigned char name[])
```

Das Modul MMC/SD bietet für den Nutzer direkt nur eine interessante Funktion:

```
unsigned char mmc_init(void)
```

Das Modul FAT bietet für den Nutzer direkt nur zwei interessante Funktionen:

```
unsigned char fat_loadFatData(void)
unsigned long long int fat_getFreeBytes(void)
```

Für die Anwendungen der Funktionen gibt es Beispiele, Kommentare und Dokumentation in den Sourcen. Ein einfaches Beispiel auch unter Code Beispiel.
Alle Funktionen, die mit FAT-Namen in Zusammenhang stehen (Datei- und Ordner-Namen), müssen folgende Konvention erfüllen:
Die FAT-Namenskonvention ist immer 8.3, das heißt, ein Datei-Name mit maximal 8 Zeichen und eine Datei-Endung mit maximal 3 Zeichen.
Ein Dateiname "test.txt" MUSS in "TEST   TXT" gewandelt werden! Noch ein Beispiel: "MAIN   C " = "main.c"
Nur vFAT erfüllt lange Datei-Namen !

Interne Technik

[Bearbeiten]

In der Regel wird das Schicht-Modell eingehalten. Das heißt, jedes Modul nutzt nur die Funktionen aus der Schicht darunter, es gibt allerdings Ausnahmen, aus Gründen der Code-Größe. Beispiel : ffwrite nutzt direkt die Funktion mmc_write_sector(sector); aus dem Modul mmc.c. Aufgrund der Namens-Konvention ist aber leicht zu erkennen, woher die Funktion kommt.

Es gibt in dem Modul fat, "Daten-Ketten". Die Aufgaben sind atomar aufgeteilt. Zum lesen von Daten sind beispielsweise die Funktionalitäten, lesen eines Sektors, auswerten der Information und weitere Entscheidung (nächsten Sektor laden usw.) nötig. Die Idee dabei ist gewesen, die Funktionen der atomaren Aufgabe nach zu implementieren.

Beispielkette:

```
fat_loadSector -> fat_loadRowOfSector -> fat_loadFileDataFromCluster -> fat_loadFileDataFromDir
```

Diese Kette wird beispielsweise beim Öffnen einer Datei durchlaufen. Um die Datei zu öffnen, muss man wissen ob es die Datei im aktuellen Verzeichnis gibt. Um das herauszufinden, muss man den ersten Sektor des Verzeichnisses laden. Dann muss man die Reihen (immer 32 Byte am Stück), also Datei/Ordner-Einträge des Sektors prüfen. Da ein Cluster aus verschiedenen Sektoren bestehen kann, müssen diese auch geprüft werden. Als letzte logische Einheit müssen jetzt noch die verketteten Cluster geprüft werden, also das ganze Verzeichnis. Ist diese Kette so durchlaufen, weiß man, ob die Datei da ist oder nicht.

Lesen

[Bearbeiten]

Beim lesen, einer Datei wird in der FAT nach verketteten Clustern gesucht. Bei einer unfragmentierten FAT liegen im Idealfall alle Cluster in einer Reihe. Dies wird ausgenutzt: Es wird der erste Cluster der Datei gesucht und solange gesucht, bis MAX_CLUSTERS_IN_ROW am Stück gefunden wurden, oder ein Abbruch der Kette erkannt wird. Wird ein Abbruch erkannt, wird erstmal das bekannte Teilstück der kompletten Kette gelesen, danach wird das nächste Teilstück gesucht. Bis die ganze Kette durchlaufen wurde.
Bei einer unfragmentierten Fat kann man oft die ganze Datei lesen ohne einen weiteren Fat lookup, das ist extrem effizient!

Schreiben

[Bearbeiten]

Beim schreiben, einer Datei wird in der FAT nach leeren Clustern gesucht. Bei einer unfragmentierten FAT liegen diese im Idealfall alle nebeneinander. Das wird ausgenutzt:

1. Es werden ab Cluster Nr. 2, MAX_CLUSTERS_IN_ROW am Stück gesucht, oder so viele, wie es freie am Stück gibt.
2. Diese werden beschrieben (also die Sektoren die mit diesen Clusternummern in Zusammenhang stehen). Sind genügend freie für die Datei Daten bekannt, werden die Cluster verkettet und man ist fertig.
3. Sind nicht genügend freie bekannt, wird die erste Teil Kette verkettet und die letzte Cluster Nummer dieser Kette gesichert (file.lastCluster). Jetzt werden wieder neue gesucht.
4. Sind jetzt immer noch nicht genügend freie Cluster bekannt, werden die beiden Teilketten verkettet (wo die alte aufhört weiß man durch file.lastCluster). Weiter bei 2.)

Bei beiden Methoden kann es zu größerem Overhead kommen, wenn die FAT fragmentiert ist. Im Extremfall ist immer nur ein Cluster am Stück frei bzw. verkettet.

Funktionsübersichts Diagramm

[Bearbeiten]

(Link) Größe: 2412 x 7946 Pixel, ältere Version der Lib. Viele der Funktionen enden in Schreiboperationen, weil das Diagramm mit der Option "MMC_OVER_WRITE TRUE" erstellt wurde.

Die Richtlinien

[Bearbeiten]

Allgemein:

Es ist immer darauf zu achten, dass es sich bei MMC/SD-Karten um Flash-Speicher handelt, dieser ist NICHT unendlich oft beschreibbar. Die Schreiben/Anhängen- und Löschen- Funktionalitäten sollten so selten wie möglich benutzt werden!
Eine SD Karte besitzt einen eigenen Controller in der Karte, der auf den eigentlichen Flash-Speicher schreibt. So können die logischen von den physikalischen Sektoren getrennt werden (Wear Leveling). Also gibt es nicht so viele Schreib Vorgänge auf dem gleichen Sektor, weil der Karten Controller den logischen Sektor auf einen anderen physikalischen schreibt.

Siehe Auch: Wikipedia SD Karten , und MMC- und SD-Karten.

Richtlinien stable-mmc-0.5.x:

Wenn eine Datei zum schreiben geöffnet ist, aber gerade keine Daten zum schreiben vorliegen, bringt es keinen Vorteil (Strom oder Geschwindigkeit) die Datei zu schießen, wenn später noch Daten geschrieben werden sollen. Bei jedem ffclose, wird der Datei Eintrag geupdatet, also dieser Sektor geschrieben (möglicherweise zusätzlich sogar ein FAT Sektor).

Oftmaliges anlegen und löschen einer Datei schreibt immer die selben (logischen) Sektoren. Besser nicht löschen und einfach neu anlegen, mit anderem Namen, oder überschreiben, da dabei die schon verketteten FAT Sektoren nicht nochmals beschrieben werden, sondern nur die Daten Sektoren überschrieben werden.

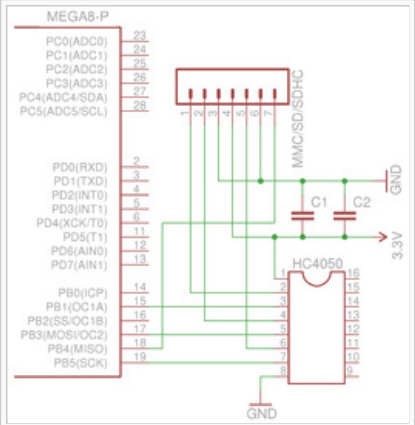
Beispiel Beschaltung

[Bearbeiten]

Schaltplan

[Bearbeiten]

MMC/SD-Karten brauchen 3,3 Volt. Wenn der Controller mit 5 Volt arbeitet, muss man die Spannungspegel anpassen. Arbeitet der Controller auch mit 3,3 Volt kann man alle Leitungen direkt mit der Karte verbinden.
Als Beispiel für die mögliche Frequenz des hier verwendeten HC4050 zur Pegelwandlung: bei +125 °C und VCC von 2 V wird das Propagationdelay mit Max 130 ns angegeben. Das macht ungefähr 7,6 Mhz Maximal mögliche Frequenz. Bis +85 °C und VCC von 2 V geht es bis 9,5 Mhz. Da aber hier 3,3 Volt anliegen wird er in allen Bereichen etwas schneller sein.
Siehe Datenblatt für mehr Infos oder Pegelwandler.



Schaltplan

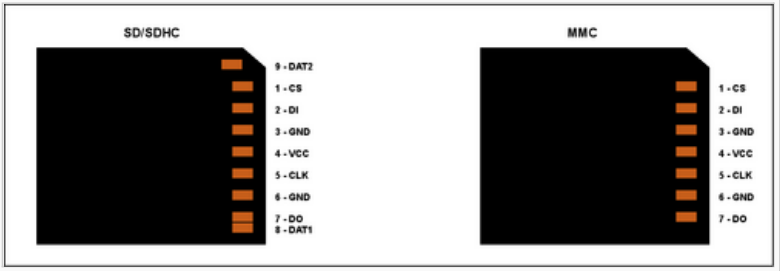
Benutzt wird der SPI-Port des Controllers (bis auf Slave Select, hier PB1). Es wäre aber auch Software basiertes SPI möglich, siehe hierfür auch Link
Die Kondensatoren C1 und C2 (100nF) dienen nur zum Entstören (möglichst nah an ICs). Sollten an jedes digitale IC in einer Schaltung.
Die MMC/SD-Leiste kann eine richtige Halterung sein oder was auch sehr gut passt ist ein Stück alte ISA-Bus Buchse eines alten Mainboards (nur MMC Karten). Die Abstände da passen gut. Einfach vom Board Flexen und dann kleine Kabel an den Stecker löten (da wo man unten jetzt das Metall sieht)
Das HC4050 IC ist ein "high to low" level shifter. Es sorgt dafür, dass die 3 abgehenden Signale des MC auf 3.3 Volt gebracht werden. Das abgehende Signal der Karte zum Controller wird auch so als logisch high erkannt.

Pinbelegung MMC/SD/SDHC

[Bearbeiten]

Pin Karte	Karte	Funktion	Pin Controller	Controller	Funktion
1	CS	Chip Select	15	PB1	Custom Slave Select
2	DI	Data In	17	MOSI (PB3)	Master Out Slave In
3	GND				
4	VCC				
5	CLK	Clock	19	SCK (PB5)	Clock
6	GND				
7	DO	Data Out	18	MISO (PB4)	Master In Slave Out

SD/SDHC/MMC Karte - Atmega8/168/328



Pins SD/SDHC/MMC Karte

Der Code

[Bearbeiten]

Sourcen [Bearbeiten]

- Sourcecode mit Linux Makefile und als AvrStudio-Projekt jetzt immer auf: SVN AVR-Fat32

Einfaches Code Beispiel [Bearbeiten]

Ein Beispiel um in eine Datei zu schreiben/anhängen und wieder zu lesen:

```
#include <stdlib.h>
#include <avr/io.h>

#include "config.h"     // Hier werden alle nötigen Konfigurationen vorgenommen !
#include "file.h"
#include "fat.h"
#include "mmc.h"                // Hardware abhängig
#include "uart.h"               // Hardware abhängig, es kann auch eine eigene eingebunden werden !


//*************************************************************************************************
void main(void){

        // Uart initialisierung zu Ausgabe von Daten
        uinit();

        uputs((unsigned char*)"\nBoot");

        // Versuch Karte zu Initialisieren, bis es klappt.
        // Umbedingt so, weil die Initialiesierung nicht immer auf anhieb klappt.
        while (FALSE==mmc_init()){
                nop();
        }

        uputs((unsigned char*)"...");


        // Fat initialisieren. Nur wenn das klappt sind weitere Aktionen sinnvoll, sonst endet das Pro
        if(TRUE == fat_loadFatData()){

          // Wenn auf dem terminal "Boot... OK" zu lesen ist, ist Init OK.
          // Jetzt kann man schreiben/anhaengen/lesen
          uputs((unsigned char*)"Ok\n");



          // Dateinamen muessen in diesem Format sein !
          // Man beachte die Größe des Arrays und die Großbuchstaben!
          unsigned char file_name[13]="TEST    TXT";

          // String zum in die Datei schreiben.
          unsigned char str[14]="Hallo Datei!";







          // Datei existiert nicht, also anlegen !
          if(MMC_FILE_NEW == ffopen(file_name)){

             // Schreibt String auf Karte !
             // Nur richtige Strings koennen mit ffwrites geschrieben werden !
             ffwrites(str);

             // Neue Zeile in der Datei
             ffwrite(0x0D);
             ffwrite(0x0A);

             // Schließt Datei
             ffclose();
           }



          // Datei existiert, also anhaengen !
```

```
        if(MMC_FILE_EXISTS == ffopen(file_name)){

          // Spult bis zum Dateiende vor um anzuhaengen, geht auch ohne Option MMC_OVER_WRITE !
          ffseek(file.length);

          // Schreibt String
          ffwrites(str);

          // Neue Zeile in der Datei
          ffwrite(0x0D);
          ffwrite(0x0A);

          // Schließt Datei
          ffclose();
        }



        // Datei existiert also lesen.
        // Gerade angelegt und beschrieben, oder war schon vorhanden und es wurde was angehaengt.
        if(MMC_FILE_EXISTS == ffopen(file_name)){

          // Setzen einer Variable und dann runterzaehlen geht am schnellsten !
          unsigned long int seek=file.length;

          // Lesen eines chars und Ausgabe des chars.
          // Solange bis komplette Datei gelesen wurde.
          do{
             uputc(ffread());
          }while(--seek);

          ffclose();
        }



      }

}
```

FAT 32 Grundlegendes

[Bearbeiten]

Ein Dateisystem des Typs FAT besteht aus mehreren Bereichen. FAT ist immer little Endian. Ein Dateisystem wie FAT32 ist ein Verwaltungsinstrument. Der FAT16/32 Standard beschreibt die Art und Weise wie die Daten Verwaltet werden.
Im Folgenden werden die Schritte die nötig sind um Daten aus dem System zu lesen und in das System zu schreiben erläutert. Die Zahlenbeispiele und Bilder sollen ein praktisches Beispiel geben. Zum Verständnis empfiehlt es sich, das ganze Kapitel FAT32 Durchzulesen !

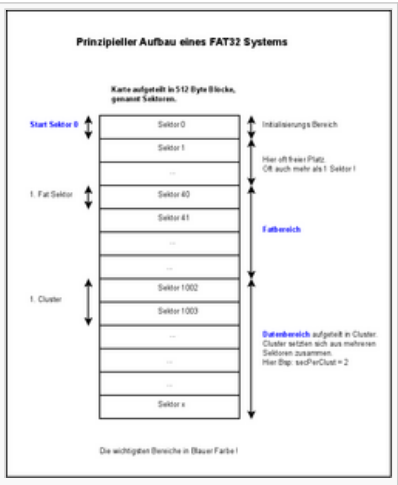


Bild 0: Grundlegende FAT32 Struktur

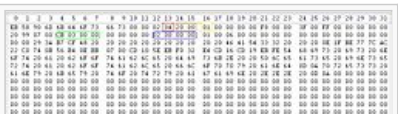
- Der erste Teil, ist der Sektor 0 oder LBA. In dem Sektor stehen die nötigen Informationen um mit dem Auslesen beginnen zu können.
- Dann die FAT selber, also die Dateizuordnungstabelle.
- Der dritte Teil ist der "Daten" Bereich, in dem beispielsweise Dateieinträge/Ordnereinträge oder Daten stehen.

FAT bedeutet File Allocation Table (Dateizuordnungstabelle). Dateizuordnungstabelle beschreibt schon das Prinzip der FAT. Kern des Dateisystems ist eine einfach verkettete Liste, in der festgehalten wird wo sich die Daten befinden, die Dateizuordnungstabelle !
In dieser Tabelle werden Cluster verkettet, diese verketteten Cluster können Dateien oder Ordner bilden, dazu später mehr.
Der Unterschied zwischen FAT32 und FAT16 besteht darin, dass ein FAT16 Dateisystem noch einen speziellen Teil hat. Das Root-Dir ist nicht einfach ein weiterer Ordner wie bei FAT32 sondern ein festgelegter Teil, der Platz für 512 Einträge bietet. FAT einträge bei FAT16 sind nur 2 Byte groß. Der Rest ist gleich.

Sektor 0 oder LBA

[Bearbeiten]

Um das Dateisystem lesen zu können muss man wissen wo was steht. Die nötigsten Informationen sind:



- Sektoren pro Cluster (rot)

- Reservierte Sektoren nach Sektor 0 (orange)
- Anzahl der FATs (gelb)
- Wieviele Sektoren von einer FAT belegt werden (grün)
- Der Root-Dir Cluster (blau)

Bild 1: Sektor 0 (512 Byte) bei einer unpartitionierten Karte

Aus diesen Daten ergibt sich demnach:

- 1. Sektor der 1. FAT ist Sektor 32 (Bild 2), also der erste Sektor nach der Anzahl der Reservierten (orange).
- Der erste Daten Cluster ist 1003, weil die Anzahl der FATs 1 ist, also einmal die Sektoren, die von einer FAT belegt werden (grün) (00003cb=971) 971 + 32 (orange) = 1003.
- Das Root-Dir ist Cluster 2, also Sektor 1003.
- Die Anzahl der Sektoren pro Cluster ist 4 (2048 Bytes/Cluster).

Damit sind alle nötigen Daten vorhanden !

Mann weiß jetzt wo das Root-Dir beginnt, ab da spannt sich der Verzeichnis Baum auf. So kann man jetzt alle Dateieinträge lesen. Es sind also alle Informationen mit der man das Lesen einer Datei beginnen kann bekannt, bzw. man weiß wo man einen Dateieintrag machen muss beim schreiben. Um eine Datei aber vollständig lesen zu können, muss man alle Cluster kennen, die zu der Datei gehören, da kommt die FAT-Tabelle ins Spiel von der man ja jetzt auch die 1. Sektornummer kennt. Dort stehen ab dem 1. Cluster der Datei, verkettet alle Folgecluster.

Bei FAT Dateisystemen können mehrere Sektoren (512 Bytes) logisch zu Clustern zusammengeschlossen werden. Ein Cluster ist für eine Datei die kleinste Speichereinheit. Das hat zur Folge, dass eine 100 Byte große Datei eventuell 2048 Bytes im Dateisystem belegt. Die rote Zahl gibt an wieviele Sektoren ein Cluster bilden. Der Daten Bereich wird immer in Clustern angegeben, das macht die Umrechnung von Sektoren zu Clustern nötig. In diesem Fall ist Cluster 2 der absolute Sektor 1003.

Die FAT

[Bearbeiten]

Folgendes Bild zeigt den ersten FAT Sektor (512 Bytes) eines FAT32 Dateisystems (nicht den ersten Sektor der Karte!). Eingetragen sind: Root-Dir und zwei darin enthaltene Ordner sowie eine Datei mit 60.000 Bytes Größe.

Die Einträge 0-7 sind immer reserviert (Quasi Cluster 0 und 1)! Also beginnt die FAT ab 8 (rot, Cluster Nummer 2). Der erste Eintrag (rot) ist 4 Byte groß und zu lesen von Stelle 8 zu Stelle 11 weil little Endian, also umgedrehte Wertigkeit. 8 niedrigste Wertigkeit, dann 9 eins höher, 10 noch eine höher und 11 höchste (dazu später noch ein Beispiel).
Wie von Microsoft empfohlen ist der erste mögliche Cluster das Root-Dir (rot). Welches hier nur einen Cluster lang ist.

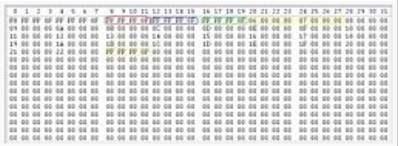


Bild 2: 1. FAT Sektor (hier Sektor 32)

An den gelben Einträgen wird die einfach verkettete Liste der FAT deutlich. Der erste gelbe Eintag steht an der Stelle des Clusters Nummer 5 und hat als Inhalt eine 6. Das bedeutet: Cluster 5 und 6 gehören zusammen. Würde an der Stelle des Clusters Nummer 5 ein FFFFFF0F stehen (little Endian) ist nur der Cluster 5 mit Daten belegt (wie bei rot, blau und grün).
So tastet man sich vom ersten Cluster einer Datei bis zum Letzten Cluster einer Datei vor. Das ist der einzige Zweck der FAT (mit FAT ist hier die eigentliche Tabelle gemeint).
Woher bekommt man aber den 1. Cluster einer Datei?

Daten

[Bearbeiten]

Ein Dateieintrag besteht aus 32 Bytes und liegt in einem Ordner/Dir im Datenbereich des Dateisystems (32 Bytes = eine Zeile).

Dies ist der 1. Cluster eines Ordners. Hier sieht man den "." bzw ".." Eintrag in Zeile 1 bzw. 2. Diese beiden Einträge sind in jedem ersten Cluster eines Ordners vorhanden. Außer im Root-Dir, von dort spannt sich der Verzeichnis Baum ja erst auf. Der "." Eintrag hat als 1. Cluster Eintrag den eigenen Cluster, hier 4. Der ".." Eintrag ist interessanter, weil er den 1. Cluster des Übergeordneten Ordners enthält, hier 3. Ist der Übergeordnete Ordner das Root-Dir, ist der 1. Cluster Eintrag 0. Diese beiden Einträge werden in der Lib genutzt um rekursiv zu löschen.

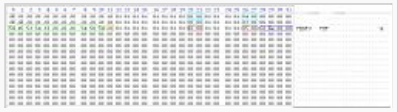


Bild 3: Inhalt eines Ordners/Dirs (hier Cluster 4)

Die dritte Zeile des Bilds zeigt eine Datei namens TEST3.TXT (grün) mit 60.000 Bytes Größe (blau) und dem 1. Cluster 5,damit ist immer der 1. Daten Cluster gemeint (rot, Folgecluster siehe Bild 2). Speziell ist hier, dass die 4 Bytes des ersten Clusters aufgeteilt sind auf 2 unterschiedliche "Orte" im 32 Byte Eintrag (Wertigkeit: 26:27:20:21).

Um nun die Daten der Datei lesen zu können, muss man die Daten Cluster kennen. Der erste Daten Cluster steht im Datei Eintrag, hier Cluster 5. Für die Folgenden sieht man in der FAT nach. Dort stehen die Folgecluster der Datei. An der Stelle des ersten Clusters der Datei Cluster 5 steht eine 6, d.h. der nächste Cluster ist 6. In dieser Form ist die FAT verkettet. Wenn in der FAT der letzte Cluster erreicht ist, inhalt des Clusters ist FFFFFFF0, liest man den letzten Sektor bis man insgesammt die Größe der Datei (blau) gelesen hat und ist fertig.

Root Dir

[Bearbeiten]

Die Sektor Nummer des Root-Dirs, bei FAT32, wird aus Sektor 0 oder dem LBA ausgelesen (siehe Bild 1). Zu Sehen ist hier in der ersten Zeile ein 32 Byte Eintrag eines Ordners. Zu erkennen ist dieser am Datei Attribut 0x10 an Stelle 11 (rot). Für den Ordner Namen ist das Gleiche Feld da wie für einen Dateinamen (grün). Der 1. Cluster des Ordners ist 3 (blau, Inhalt des Ordners ist Bild 3). Bei Ordnern werden die Felder für die Größe genullt (orange). Das Root-Dir hat keinen "." bzw. ".." Eintrag. Der erste Cluster des Dirs ist ja über Sektor 0 bekannt und ab dem Root-Dir spannt sich der Verzeichnisbaum erst auf. Das Root-Dir bei FAT32 kann beliebig groß werden.

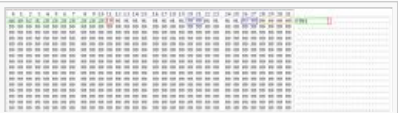


Bild 4: Root des Dateisystems (hier Cluster 2)

Zusammenfassung

[Bearbeiten]

Für Dateien und Ordner sind nur zwei Bereiche eines FAT32 Dateisystems wichtig, nämlich die FAT selbst und der Daten Bereich. Eine Datei wird aufgesplittet, in einen Datei Eintrag und in die Nutzdaten der Datei (alles im Daten Bereich). Ein Ordner ist vom Eintrag im Dateisystem einer Datei sehr ähnlich. Wo bei einer Datei über den 1.Cluster eine Cluster-Chain für die eigentlichen Daten der Datei beginnt, wird bei einem Ordner so eine Cluster-Chain für weitere Datei/Ordner Einträge verkettet. Ein Ordner bietet pro Sektor maximal 16 Einträge (16*32=512).

Siehe auch

[Bearbeiten]

- Thread: http://www.mikrocontroller.net/topic/105869
- Infos (englisch): https://www.pjrc.com/tech/8051/ide/fat32.html
- Fatgen103 von Microsoft (einfach mal nach Fatgen103.pdf suchen..)
- MMC- und SD-Karten

Weblinks

[Bearbeiten]

Weitere Projekte mit FAT MMC/SD Karten:

- MMC/SD FAT16 card reader example application (Roland Riegel)
- MMC/SD FAT16 (Ulrich Radig)
- MMC/SD FAT16/32 auch multi File (Holger Klabunde)

Kategorien: AVR | Speicher und Dateisysteme