Benutzen einer SD-Speicherkarte mit dem ATmega-Microcontroller

Simeon Maxein

27. Februar 2008

Inhaltsverzeichnis

1 Einführung

1.1 Was ist eine SD-Karte?

Abbildung 1: Vorder- und Rückseite einer SD-Karte.
Foto von Andreas Frank,
benutzt unter der Creative Commons Attribution 2.5 License

Wenn hier von einer SD-Karte gesprochen wird, dann ist eine SD-Speicherkarte gemeint. Es gibt auch sogenannte SDIO-Karten, die zum Anschluss von Geräten wie z.B. Digitalkameras benutzt werden, aber darauf will ich in dieser Ausarbeitung nicht weiter eingehen.

Eine SD-Karte ist eine Speicherkarte mit zusätzlichen Funktionen für DRM-Anwendungen. Diese DRM-Funktionen können benutzt werden, um geschützte Karteninhalte nur einem bestimmten Gerät zugänglich zu machen, damit z.B. eine Videodatei, die auf der Karte abgelegt ist, nur auf einem bestimmten DVD-Player abgespielt werden kann. Damit soll verhindert werden, dass der Film einfach weiterkopiert wird. Diese Funktionen sind aber ebenfalls nicht von Interesse für unser Projekt, und die Spezifikationen für diese Features sind nicht frei verfügbar.

Die Datenspeicherung übernimmt auf der Karte meistens ein NAND-Flash-Chip, das ist jedoch nicht festgelegt: In der Spezifikation sind auch Karten mit gewöhnlichen ROMs (nicht beschreibbar), PROMs (einmal beschreibbar) oder RAM-Chips erwähnt. Neben dem Speicherchip sitzt noch ein Controller, der dafür zuständig ist die Daten über die Pins der Karte einzulesen oder auszugeben, und in geeigneter Weise an den Speicherchip weiterzugeben.

Das ist nicht so trivial wie es sich erstmal anhört, z.B. benutzen viele SD-Karten so genannte Wear-Levelling Techniken, die die Lebensdauer der Karte erhöhen sollen. Ein Flash-Chip hält nur eine begrenzte Anzahl an Schreibzyklen aus. Wenn man immer wieder Daten auf die gleiche Speicheradresse auf einem Flash-Chip schreibt, ist diese Stelle relativ schnell defekt (normalerweise nach über 100 000 Schreibzyklen, aber das kann man innerhalb von Sekunden erreichen). Deshalb verteilen viele (laut Wikipedia alle) Speicherkarten Schreibzugriffe zum gleichen „logischen“ Sektor auf verschiedene physikalische Sektoren auf dem Speicherchip. Wir müssen uns also vermutlich erstmal keine Gedanken darüber machen, die Karte zu zerschreiben.

Der Controller-Chip kann außerdem Prüfsummen berechnen, um die Korrektheit der

Daten auf der Karte und bei der Übertragung festzustellen, und kann Auskunft über die Eigenschaften der Karte geben, z.B. Speicherkapazität, Hersteller und Stromverbrauch. Den nötigen Speicherplatz für alle diese Aufgaben liefert bei Flash-Karten normalerweise der Flash-Chip selber, weshalb auf einer solchen SD-Karte meistens etwas weniger Platz zur Verfügung steht als erwartet; meine „128 Megabyte“ SD-Karte stellt zum Beispiel nur ca. 120 MiB zur Verfügung.

1.2 Was ist der Unterschied zwischen MMC- und SD-Karten?

Der SD-Standard wurde 2001 entwickelt, wobei der MMC-Standard als Vorlage diente. Tatsächlich stimmen die Spezifikationen in einigen Teilen fast wörtlich überein. Die beiden Kartentypen haben ähnliche Abmessungen, wobei SD-Karten etwas dicker sind damit sie nicht in MMC-Slots passen. Fast alle SD-Kartenleser können auch mit MMC-Karten umgehen, in die andere Richtung funktioniert das jedoch nicht.

SD-Karten besitzen zwei zusätzliche elektrische Kontakte (9 gegenüber 7 bei den MMCs), die links und rechts neben die Kontakte der MMC gequetscht wurden. Dadurch (und durch veränderte Benutzung eines der bestehenden Kontakte) wurde es möglich, den Datenbus von 1 auf 4 Leitungen zu vergrößern, um schnellere Übertragungen bei gleicher Busgeschwindigkeit zu erlauben. Es gibt inzwischen auch Weiterentwicklungen der MMC, z.B. die MMCplus, die zusätzlich zu den Kontakten der SD-Karte noch eine zweite Reihe von Kontakten eröffnet hat, und damit einen 8-bit Datenbus bietet.

1.3 Anwendungsbeispiele

SD-Karte am ATmega, wozu kann man das eigentlich brauchen? Hier sind ein paar Vorschläge:

- zum Aufzeichnen von Messdaten, z.B. um später am PC Kurven anzeigen zu können
- Abspielen von Musikdateien von der Karte, oder Aufzeichnen von Sound
- Anzeige von Texten
- Datenschredder zum sicheren Löschen aller Daten auf der Karte
- Als „Auslagerungsspeicher“, falls man speicherintensive Berechnungen durchführen muss (dafür wären externe RAM-Bausteine jedoch besser geeignert)

2 Verbinden der Hardware

2.1 Pin-Belegung

Wie man die Karte anschließen muss, hängt stark davon ab, mit welchem Übertragungsprotokoll man die Karte benutzen will; die vorhandenen Möglichkeiten werden im Abschnitt 3.1 genauer beschrieben. Hier bietet sich der SPI-Modus an, da der ATmega

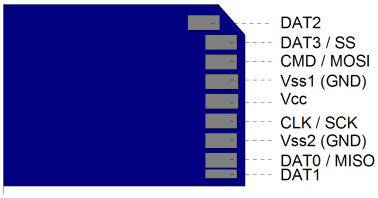

Abbildung 2: Belegung der Kontakte einer SD-Karte

über eine SPI-Schnittstelle verfügt, mit der die Übertragung schnell und verhältnismäßig einfach umzusetzen ist. Will man also die Hardware-SPI-Schnittstelle des ATmega nutzen, muss man die Kontakte MISO, MOSI und SCK an die gleich benannten Pins des ATmega verbinden. Der $\overline{SS}$-Kontakt der Karte kann an einen beliebigen I/O-Pin des Controllers verbunden werden. Der Anschluss der Versorgungsspannung (Vss und Vcc) ist im Abschnitt 2.3 besprochen. Die Kontakte DAT1 und DAT2 werden im SPI-Modus nicht gebraucht und können unverbunden bleiben.

Wenn man auf die SPI-Schnittstelle des ATmega verzichten will, z.B. falls das SPI schon anderweitig gebraucht wird, kann man das Protokoll auch "Per Handümsetzen, wie in Abschnitt 3.3 beschrieben. Diese Option ist dafür deutlich langsamer und etwas mehr Programmierarbeit. MISO, MOSI, $\overline{SS}$ und SCK können dafür an beliebige I/O-Pins am Controller angeschlossen werden.

Schließlich kann man die SD-Karte auch im nativen SD-Modus betreiben; das ist jedoch nicht empfehlenswert, da die Umsetzung mit dem ATmega sowohl komplizierter als auch langsamer wäre als im SPI-Modus. Ein Grund dafür ist, dass im SD-Modus vom Controller Prüfsummen für die übertragenen Kommandos und Daten berechnet werden müssen, was viel Rechenzeit in Anspruch nehmen würde. In dieser Ausarbeitung wird deshalb davon ausgegangen, das der SPI-Modus benutzt wird.

2.2 Kartenhalterung

Es empfiehlt sich, eine Kartenhalterung zu kaufen oder selber zu basteln. Man kann natürlich auch direkt Kabel an den Pins der Karte festlöten, aber für viele Anwendungen ist es sinnvoll, die Karte tauschen bzw. entfernen zu können. Außerdem wird ein Feature der SD-Karte nur durch den Kartenhalter zur Verfügung gestellt: Der Schreibschutz-Pin ist nicht elektrisch mit dem inneren der Karte verbunden, sondern wird nur über einen Tastschalter in der Halterung überprüft. Es ist Aufgabe des Host-Gerätes, die Schalterstellung zu prüfen und gegebenenfalls nicht auf die Karte zu schreiben.

2.3 Problem: unterschiedliche Spannungspegel

Die ATmega-Mikrocontroller arbeiten normalerweise mit 5V Versorgungsspannung und entsprechend hohen Pegeln an den I/O-Pins. Eine SD-Karte verträgt aber nach Spezifikation nur 2,7V - 3,6V für Versorgung und I/O. Man muss sich also überlegen, wie man mit diesen unterschiedlichen Spannungen arbeiten kann.

2.3.1 Lösung durch Low-Voltage-ATmega

Atmel bietet in der ATmega-Reihe auch Low-Voltage-Chips an, die mit Spannungen ab 2,7V arbeiten können. Damit würden Controller und Karte von vorne herein die gleichen Spannungen benutzen, und man kann die Pins direkt verbinden. Nachteil ist jedoch, dass diese Chips nur halb so schnell sind wie die normalen 5V-Chips.

2.3.2 Lösung durch Spannungsteiler oder Dioden

Um die Spannung an der Karte zu senken, kann man einen Spannungsteiler vor den Versorgungspin der Karte schalten, besser noch vor alle Pins, die an der SD-Karte als Eingang benutzt werden. Für die MISO-Leitung müsste die Spannung zum Controller hin angehoben werden, das ist aber voraussichtlich nicht nötig. Nach dem Datenblatt für den ATmega8 wird bereits eine Spannung ab 1,8V an einem normalen I/O-Pin als 1 gewertet (bei 5V Versorgungsspannung), daher sollten die Daten auf dieser Leitung auch ohne Spannungsanhebung problemlos empfangen werden.

Alternativ kann man zum Senken der Versorgungsspannung auch Dioden benutzen. An einer 1N4148-Diode (das sind so ziemlich die billigsten die es gibt) fallen, je nach Temperatur und Stromfluss, zwischen 0,4 und 0,8 Volt ab. Wenn man drei davon hintereinander schaltet, kommt man so auf 2,6V-3,8V. Am bestem man misst einfach nach, bevor man die Karte anschließt. Nimmt man eine Leuchtdiode statt einer Diode, bekommt man zusätzich eine Art Äktivitätsanzeige", da die LED heller leuchtet, wenn die Karte mehr Strom braucht (vor Allem beim Schreiben).

2.3.3 Lösung durch Pegelwandler

Für das Umsetzen der Spannungspegel an den Datenleitungen gibt es auch spezielle Mikrochips, sogenannte Pegelwandler. Diese können die Pegel in beide Richtungen anpassen. Dazu wird erkannt, ob auf der Seite des Senders ein High- oder Low-Pegel vorliegt, der dann in einen passenden Pegel der Empfängerseite umgewandelt wird. Bei manchen dieser Chips muss man festlegen, in welche Richtung die Übertragung auf jeder Leitung läuft. Andere können das automatisch feststellen, da die sendende Seite meistens einen gewissen Strom liefern kann, während der Empfänger oft nur über einen größeren Pullup- oder Pulldown-Widerstand den Leitungszustand beeinflusst.

2.3.4 „Lösung" durch Verletzen der Spezifikation

Der ATmega8 braucht laut Datenblatt 4,5V, um zu laufen. Meistens reicht aber auch schon weniger; in meinem Versuchsaufbau lief der ATmega8 auch noch bei knapp unter 4V. SD-Karten brauchen nach der Spezifikation 2,7V-3,6V. Das wird mit 4V meistens auch noch gut gehen (meine Karte hat es auch schon bei 5V ausgehalten). Wenn man Karte und Controller auf diese Weise direkt verbindet, spart man sich zwar etwas Arbeit mit der Hardware, dafür kann es aber dann vorkommen, dass die Speicherkarte Schaden nimmt. Deshalb ist generell von dieser Lösung abzuraten.

3 Übertragungsprotokoll

Etwas vereinfacht gesagt: Um mit einer SD-Karte zu arbeiten schickt man unterschiedliche Kommandos an die Karte und erhält Antworten. Eventuell sendet oder empfängt man danach ein oder mehrere Datenpakete. In diesem Abschnitt geht es darum, wie das Senden und Empfangen überhaupt abläuft. Im nächsten Abschnitt werden dann die Kommandos erklärt, die nötig sind, um die Karte zu initialisieren und Daten zu empfangen und senden.

3.1 Übertragungsmodi der SD-Karte

Generell kommunizieren SD-Karten über das SD-Bus-Protokoll. Die Übertragung läuft hier auf einer „Command"-Leitung (auf der die Karte Kommandos entgegennimmt und gegebenenfalls darauf antwortet), und wahlweise einer oder vier Datenleitungen (auf denen die Datenpakete übertragen und bestätigt werden). Alle Kommandos und Datenpakete enthalten eine CRC-Prüfsumme, um eine korrekte Übertragung sicherzustellen. CRC-Prüfsummen sind jedoch aufwändig zu berechnen. Bei gelesenen Daten und Kommando-Antworten von der Karte könnte man die Prüfsumme einfach ignorieren, aber wenn die Karte ein Kommando oder Datenpaket mit falscher CRC bekommt, wird dieses abgelehnt, und eine Fehlermeldung wird zurückgesendet. Außerdem werden manche Kommandos nicht immer beantwortet, sondern man muss auf einen Timeout warten. Es gibt jedoch ein alternatives Protokoll, das von allen SD-Karten mit Flash-Speicher unterstützt wird.

SPI (Serial Peripheral Interface) ist ein von Motorola entwickelter Standard zum Anschluss von Peripheriegeräten (Slaves) an einen „Master". Dieses Protokoll ist einfacher zu unterstützen, da hier z.B. jedes Kommando immer eine Antwort erzeugt, und die Karte CRC-Prüfsummen generell ignoriert (man kann die Überprüfung aber bei Bedarf auch einschalten). Ein weiterer Vorteil ist, dass auch MMC-Karten den SPI-Modus unterstützen, man muss allerdings auf ein paar Unterschiede bei den unterstützten Kommandos achten. Für den Rest dieser Ausarbeitung wird davon ausgegangen, dass die SD-Karte im SPI-Modus betrieben werden soll.

Zum Anschließen eines einzelnen Slaves, in diesem Fall unsere SD-Karte, werden insgesamt vier Leitungen benötigt:

- $\overline{SS}$ (Slave select): Mit dieser Leitung wird ausgewählt, welcher Slave angesprochen wird, indem diese Leitung auf low gezogen wird.
- MISO (Master in, Slave out): Datenleitung, die vom Master an alle Slaves verbunden ist. Hier sendet der angesprochene Slave Daten an den Master.
- MOSI (Master out, Slave in): Wie oben, aber hier werden Daten vom Master an den angesprochenen Slave übertragen.
- SCK (Serial Clock): Ebenfalls an den Master und alle Slaves angeschlossen, hier gibt der Master den Takt der Übertragung an. Ein Slave kann nicht von sich aus

Daten senden, sondern nur, wenn bei ihm $\overline{SS}$ auf low gezogen wurde und der Master ein Taktsignal vorgibt.

Der ATmega verfügt über eine SPI-Schnittstelle, wodurch erstmal die Arbeit erleichtert wird, da man sich nicht selbst um die Einzelheiten des Übertragungsprotokolls kümmern muss. Da jedoch bei unserem Einchip-Computer die SPI-Schnittstelle bereits für die Ausgabe des Videosignals gebraucht wird, ist es später nötig das SPI-Protokoll „per Hand“ umzusetzen - Das ist jedoch nicht zu kompliziert, unter Anderem da eine genaue Einhaltung eines Taktes nicht erforderlich ist. Der Mikrocontroller gibt als Master schließlich selbst den Takt vor, und nach Spezifikation der SD-Karte darf dieser beliebig langsam sein. Man könnte also eigentlich die SD-Karte auch mit Stift, Papier, drei Kippschaltern und einer Leuchtdiode betreiben.

SPI ist ein relativ simples Protokoll. Um Daten auszutauschen, zieht der Master zuerst die $\overline{SS}$-Leitung des gewünschten Slave-Geräts auf Low. Dann erzeugt der Master ein Clock-Signal auf der SCK-Leitung. Mit jeder Clock wird über die MOSI-Leitung ein Bit vom Master zum Slave übertragen, und über die MISO-Leitung ein Bit vom Slave zum Master. Wenn die Übertragung abgeschlossen ist, stoppt der Master das Clock-Signal und zieht $\overline{SS}$ wieder auf High.

Die Übertragung läuft also immer in beide Richtungen gleichzeitig. Beim Ansprechen der SD-Karte sendet aber meistens nur entweder der Master oder der Slave sinnvolle Daten. Wenn einer von den beiden nichts wichtiges mitzuteilen hat, zieht er die entsprechende Datenleitung auf high, er sendet also nur Einsen.

Im SPI-Protokoll ist nicht genau festgelegt, an welcher Flanke des Taktsignals die Daten auf den Bus gelegt werden sollen. Es gibt dazu vier verschiedene Einstellungen, auch SPI-Modi genannt. SD- und MMC-Karten benutzen Modus 0 (Clock beginnt Low, Daten werden bei der ersten Flanke übernommen).

Das Protokoll zur Unterhaltung mit der SD-Karte ist Byte-orientiert (ich bin mir nicht sicher ob das bei allen SPI-Variationen der Fall ist). Das heißt, dass die Daten auf dem Bus immer in Blöcken zu 8 Bit behandelt werden, und eine Byte beginnt auch immer bei einem Vielfachen von 8 Clocks nachdem die $\overline{SS}$-Leitung auf Low gezogen wurde. Dabei wird das höchstwertige Bit zuerst übertragen.

3.2 Benutzen der SPI-Schnittstelle am ATmega

3.2.1 Initialisierung

Zuerst sollte man im Datenrichtungsregister des entsprechenden I/O-Ports die Leitungen MOSI, $\overline{SS}$ und SCK als Ausgang, und MISO als Eingang festlegen. Dann kann man die SPI-Schnittstelle konfigurieren, indem man die richtigen Einstellungen in das SPCR (SPI Control Register) und SPSR (SPI Status Register) schreibt. Im SPCR sind dabei folgende Bits für uns von Interesse:

- Bit 7 - SPIE: SPI Interrupt Enable

  Sollte selbsterklärend sein.

- Bit 6 - SPE: SPI Enable

  Auf 1 setzen.

- Bit 5 - DORD: Data Order

  Auf 0 setzen, damit die Daten MSB first empfangen/gesendet werden.

- Bit 4 - MSTR: Master/Slave select

  Auf 1 setzen, weil der uC bei der Übertragung als Master arbeiten muss.

- Bit 3 - CPOL: Clock Polarity

  Dieses Bit bestimmt, ob die erste Taktflanke eine fallende oder eine steigende Flanke ist, und damit auch ob SCK zwischen den Übertragungen High oder Low ist. Auf 0 setzen, dann ist SCK Idle-Low.

- Bit 2 - CPHA: Clock Phase

  Hier wird festgelegt, ob die Daten bei der ersten oder zweiten Taktflanke übernommen werden sollen. Auf 0 setzen, da von der SD-Karte bereits die erste Flanke benutzt wird.

- Bits 1 und 0 - SPR1 und SPR0: SPI Clock Rate select

  Über diese Bits kann man die Geschwindigkeit der Schnittstelle festlegen, abhängig vom Systemtakt und dem SPI2X-Bit im SPSR. Die Einstellungsmöglichkeiten kann man im ATmega-Datenblatt nachlesen, ich setze beide Bits auf 0, um die maximale Geschwindigkeit einzustellen.

Im SPSR ist erstmal nur Bit 0 (SPI2X) für uns interessant, das die Clock-Geschwindigkeit um den Faktor 2 beeinflusst. Für die größtmögliche Geschwindigkeit sollte man das Bit auf 1 setzen. Mit SPR1 und SPR0 auf 0, und SPI2X auf 1, arbeitet die SPI-Schnittstelle mit halbem Systemtakt - also bei ATmegas bis zu 10Mhz. Das ist für die Speicherkarte kein Problem, eine MMC muss bis 20Mhz, eine SD-Karte bis 25Mhz arbeiten können.

Beispiel C-Code für die Initialisierung:

```
SPCR = (1<<SPE)|(1<<MSTR);
SPSR = (1<<SPI2X);
```

3.2.2 Bytes senden und empfangen

Da SPI bidirektional arbeitet, wird immer ein Byte gleichzeitig gesendet und empfangen. Um einen Datentransfer zu starten, schreibt man das zu sendende Byte in das SPI Data Register (SPDR). Wenn die Übertragung abgeschlossen ist, setzt die Schnittstelle das SPI Interrupt Flag (SPIF) im SPSR. Wenn man dieses Flag überprüft und so das Ende der Übertragung festgestellt hat, kann man danach aus dem SPDR das empfangene Byte lesen. Dabei wird das SPIF automatisch wieder gelöscht.

Wenn man nur ein Byte lesen will, weil man z.B. auf eine Antwort der Karte wartet, sendet man einfach ein 0xff. Die Karte macht das gleiche, wenn sie nur Daten empfängt. Wenn man nur senden will, kann man das empfangene Byte einfach ignorieren.

Damit das Byte auch wirklich von der Karte empfangen wird, muss $\overline{SS}$ auf Low gezogen sein. $\overline{SS}$ soll generell während einer Transaktion mit der Karte auf Low-Pegel bleiben, und danach wieder auf High gezogen werden. Siehe dazu den Abschnitt 3.4.1.

3.3 SPI-Protokoll „zu Fuß“ umsetzen

3.3.1 Initialisierung

Auch hier müssen zuerst die entsprechenden Bits in den DDRs gesetzt / gelöscht werden, die zu den I/O-Pins gehören an denen die Karte angeschlossen ist. MOSI, $\overline{SS}$ und SCK werden als Ausgang benutzt, MISO als Eingang.

3.3.2 Bytes senden und empfangen

Hier ist die allgemeine Routine, um gleichzeitig ein Byte zu senden und zu empfangen. Zum optimieren kann man zwei getrennte Routinen je zum lesen und schreiben benutzen, da bei der Kommunikation mit der SD-Karte nicht gleichzeitig gesendet und empfangen werden muss.

- Erstes zu sendende Bit (MSB) auf die MOSI-Leitung legen
- SCK auf High ziehen
- Erstes empfangene Bit (MSB) von der MISO-Leitung lesen
- SCK auf Low ziehen
- Zweites zu sendende Bit auf MOSI legen
- SCK auf High ziehen
- ...
- Achtes empfangene Bit (LSB) von MISO lesen
- SCK auf Low ziehen

Es folgt ein Codebeispiel für das Lesen von 8 Bit. Es wird davon ausgegangen, dass sowohl SCK als auch MISO am I/O-Port B angeschlossen sind. Die Labels SPI_Clock und SPI_MISO stehen für die Pinnummer der SCK und MISO-Leitung am Port. Dabei wird die Variable inbyte mit dem Ergebnis gefüllt, indem das aktuell gelesene Bit immer in das Bit 0 von inbyte übertragen wird. Bevor das nächste Bit gelesen wird, wird inbyte um eine Stelle nach links verschoben, so dass am Ende das zuerst gelesene Bit auf die Position des MSB vorgerückt ist.

```
unsigned char inbyte;
for (U08 a=8; a>0; a--) {
  inbyte <<= 1;                          //inbyte nach links schieben.
  PORTB |= (1<<SPI_Clock);               //SCK auf High ziehen.
  if (bit_is_set(PORTB,SPI_MISO) > 0) { //Wenn MISO 1 ist:
    inbyte++;                            //LSB von inbyte auf 1 setzen
  }
  PORTB &=~(1<<SPI_Clock);               //SCK auf Low ziehen
}
```

3.4 Kommunikation mit der Karte

3.4.1 Transaktionen

Eine Transaktion mit der Karte wird immer damit begonnen, dass die $\overline{SS}$-Leitung auf Low gezogen wird. Darauf sendet der uC ein Kommando an die Karte, die nach kurzer Zeit antwortet. Je nach Art des Kommandos folgten noch Datenpakete oder sonstige Informationen. Erst wenn die komplette Transaktion beendet ist, sollte der $\overline{SS}$-Pin wieder auf High gezogen werden. Da eine Transaktion immer mit einem Kommando an die Karte anfängt, und wir nicht daran interessiert sind, mit mehreren SPI-Geräten gleichzeitig zu arbeiten, reicht es hier aus, wenn man einfach vor jedem Kommando $\overline{SS}$ kurz auf High setzt, der Karte ein paar Clocks schickt um Zeit zu vertrödeln (z.B. indem man ein Byte vom SPI-Bus liest), und dann $\overline{SS}$ wieder Low zieht, bevor das Kommando gesendet wird.

3.4.2 Kommandos

Byte	*Bedeutung*
1	Beginnt mit den Bits ”01”, danach folgt der Kommando-Index
2-5	Parameter
6	CRC7, letztes Bit immer 1

Tabelle 1: Kommando-Format

Ein Kommando an die Karte besteht im SPI-Modus immer aus 6 Bytes.

Das erste Byte enthält den Index des Kommandos, allerdings sind die ersten beiden Bits fest als ”01”vorgegeben. Es gibt also bis zu 64 verschiedene Befehle, allerdings sind die meisten für uns uninteressant. Kommandos werden mit CMD<index> abgekürzt. CMD17 (READ_SINGLE_BLOCK)ist also das Kommando, das mit dem Byte 0x51 beginnt (17 = 0x11; 0x11+ 0x40 = 0x51).

Auf das Kommando-Byte folgen vier Bytes, die eventuelle Parameter für den Befehl enthalten. Parameter, die sich über mehrere Bytes erstrecken (z.B. die Adresse bei einem Lesebefehl) werden mit dem höchstwertigen Byte zuerst gesendet.

Die CRC-Prüfsumme wird im SPI-Modus nicht von der Karte geprüft, mit zwei Ausnahmen: Bei CMD0 und bei CMD8 muss die Prüfsumme stimmen. CMD0 wird als erstes Kommando an die Karte gesendet, um diese in den Idle-Mode zu versetzen. Da der Befehl keine Parameter hat, kann man einfach eine vorberechnete Prüfsumme benutzen (siehe Abschnitt 4.1.2). CMD8 wird nur zum Initialisieren von SDHC-Karten mit einer Kapazität von 4GB und mehr gebraucht, und wird in dieser Ausarbeitung nicht besprochen.

3.4.3 Antworten

Bit	*Bedeutung*
7	Immer 0
6	parameter error
5	address error
4	erase sequence error
3	communication crc error
2	illegal command
1	erase reset
0	in idle state

Tabelle 2: Antwort-Format R1

Nachdem man das Kommando an die Karte gesendet hat, muss man auf die Antwort der Karte warten. Generell zieht die Karte die MISO-Leitung immer auf High, wenn sie nichts zu sagen hat, d.h. man empfängt immer 0xff. Um also die Antwort abzuwarten, liest man so lange Bytes von der SPI-Schnittstelle, bis man ein Byte empfängt das nicht 0xff lautet. Aus diesem Gund ist alles was die Karte sendet so ausgelegt, dass das erste Byte nie 0xff sein kann.

Alle hier besprochenen Befehle senden eine Antwort im Format R1. Das ist ein einzelnes Byte, bei dem jedes Bit für einen bestimmten Fehler oder Status steht.

Empfängt man also die Antwort 0x00 oder 0x01, wurde der Befehl ohne Fehler akzeptiert. Dabei kommt die Antwort 0x01 nur beim Initialisieren der Karte vor. Oft ist mit der Antwort die Transaktion beendet und $\overline{SS}$ kann wieder auf High gezogen werden, bei Datenlese- und -schreibbefehlen folgt jetzt noch ein Datenpaket.

3.4.4 Anwendungsspezifische Kommandos

CMD55 hat eine besondere Funktion. Wenn man der Karte ein CMD55 schickt (dazu gehört auch das Abwarten der Antwort) und danach einen weiteren Befehl, wird dieser als „Anwendungsspezifisches“ Kommando interpretiert (kurz ACMD<index>). Das ermöglicht weitere 64 Befehle, von denen wir jedoch nur einen brauchen.

3.4.5 Datenpakete

Beim lesen und schreiben von Benutzerdaten werden immer Datenpakete ausgetauscht. Datenpakete von der Karte und vom Mikrocontroller sind dabei gleich aufgebaut. Ein Paket beginnt mit einem speziellen Byte, dem sog. Data Token, das dem Empfänger den Beginn des Pakets signalisiert. Darauf folgt ein Block mit den Nutzdaten, an den zwei Bytes mit CRC-Prüfsumme angehängt sind.

Bei den einfachen Lese- und Schreibbefehlen für einzelne Blöcke lautet das Data Token immer 0xfe. Die CRC-Prüfsummenbytes werden von der Karte im SPI-Modus ignoriert, müssen aber trotzdem gesendet bzw. empfangen werden.

Wenn von der SD-Karte Daten erwartet werden, kann diese an Stelle eines Datenpaketes auch ein Error Token senden, auf das dann keine Daten folgen. Das Error Token beginnt mit drei Nullbits, die restlichen Bits geben Auskunft über die Art des Fehlers.

4 Initialisierung und simpler Datentransfer

4.1 Initialisieren der Speicherkarte

Hier gehe ich erstmal nur auf die einfache Variante der Initialisierung ein, die für die meisten SD-Karten und die MMC-Karten funktioniert.

4.1.1 Initialization Delay

Die Speicherkarte kann nicht sofort auf Kommandos reagieren, sobald sie Strom bekommt. Nach der Spezifikation soll man wenigstens 1ms warten, danach $\overline{SS}$ und MOSI auf High ziehen und wenigstens 74 Clock-Signale an die Karte senden, bevor man das erste Kommando schickt.

4.1.2 Karte in den Idle-Modus versetzen (CMD0)

Als erstes muss man der Karte das Kommando CMD0 (GO_IDLE_STATE) senden. Hier ist ausnahmsweise die CRC-Prüfsumme doch von Bedeutung, aber da der Befehl keine Parameter hat kann diese als Konstante gespeichert werden. Es empfiehlt sich für die Parameter Nullbytes zu senden, da man dann die unten angegebene CRC benutzen kann.

Die SD-Karte prüft beim Empfang von CMD0 den Pegel der $\overline{SS}$-Leitung. Ist diese auf Low gezogen (wie bei einer normalen SPI-Übertragung vorgesehen), startet die Karte in den SPI-Modus, in dem sie bleibt bis die Stromzufuhr unterbrochen wird. Ist die $\overline{SS}$-Leitung an dieser Stelle High, wird die Karte in den SD-Bus-Modus starten.

Als Antwort auf dieses Kommando sollte man ein Byte im R1-Format empfangen, das anzeigt, dass die Karte sich jetzt im Idle-Modus befindet.

```
Beispiel:
Send: 0x40 0x00 0x00 0x00 0x00 0x95 (CMD0)
Recv: 0x01 (R1-Antwort mit dem "In Idle State"-Bit gesetzt.)
```

4.1.3 Karte initialisieren (CMD1/ACMD41)

Nun muss die Karte vom Idle-Modus in den Betriebsmodus versetzt werden. Dazu hat man zwei Kommandos zur Auswahl, nämlich CMD1 (SEND_OP_COND) und ACMD41 (SD_SEND_OP_COND). CMD1 funktioniert bei allen MMCs und allen „dicken“ SD-Karten. Für die dünnen (1,4mm) SD-Karten ist CMD1 ein illegales Kommando. ACMD41 dagegen funktioniert bei allen SD-Karten, ist aber den MMC-Karten unbekannt. In der SD-Spezifikation wird empfohlen, dies als Unterscheidungsmerkmal zu benutzen, indem zuerst versucht wird, die Karte mit ACMD41 zu initialisieren. Falls die Karte darauf mit einem Fehler antwortet (insbesondere 0x05, Illegal Command), sollte man es mit CMD1 versuchen. Klappt es dann, hat man vermutlich eine MMC im Slot.

Beide Kommandos erhalten nur einen Parameter, mit dem angezeigt wird, ob der Host High-Capacity SD-Karten unterstützt. Da das aber hier nicht der Fall ist (bei SDHC-Karten muss man einige Besonderheiten beachten), senden wir wieder für die Parameter-Bytes nur Nullen. Da die CRC bei allen Kommandos (ausser CMD0 und CMD8) ignoriert wird, sollten wir nur darauf achten, dass das letzte Bit 1 ist.

Die Karte kann eine Weile brauchen, um sich zu initialisieren. Solange das noch nicht abgeschlossen ist, antwortet die Karte mit einem „0x01“, um anzuzeigen dass sie sich immer noch im Idle-Modus befindet. In diesem Fall muss man das Kommando erneut schicken, bis irgendwann ein „0x00“ empfangen wird. Damit sind wir nicht mehr im Idle-Mode und die Initialisierung ist abgeschlossen. Man sollte allerdings einen Timeout in diese Schleife einbauen, da z.B. SDHC-Karten dauerhaft mit „0x01“ antworten werden, wenn der Host diese Karten nicht unterstützt.

```
Beispiel-Initialisierung mit CMD1:
Send: 0x41 0x00 0x00 0x00 0x00 0x01 (CMD1)
Recv: 0x01 (R1-Antwort mit dem "In Idle State"-Bit gesetzt.)

(Das wird mehrmals wiederholt)
...

Send: 0x41 0x00 0x00 0x00 0x00 0x01
Recv: 0x00 (R1-Antwort, jetzt ohne "In Idle State")

Beispiel-Initialisierung mit ACMD41:
Send: 0x77 0x00 0x00 0x00 0x00 0x01 (CMD55, das "Prefix" für ein ACMD)
Recv: 0x01 (R1-Antwort mit dem "In Idle State"-Bit gesetzt.)
Send: 0x69 0x00 0x00 0x00 0x00 0x01 (ACMD41)
Recv: 0x01

(Das wird mehrmals wiederholt)
...

Send: 0x77 0x00 0x00 0x00 0x00 0x01
```

```
Recv: 0x01 oder 0x00
Send: 0x69 0x00 0x00 0x00 0x00 0x01
Recv: 0x00 (R1-Antwort, jetzt ohne "In Idle State")
```

4.2 Datentransfer

Beim Datentransfer werden immer Datenblöcke einer bestimmten Größe übertragen. Man kann die Blockgröße über ein Kommando verändern, wobei allerdings nicht jede Karte auf beliebige Größen gestellt werden kann, und für die Schreibblockgröße nochmal andere Regeln gelten als für die Leseblockgröße. Da die Angelegenheit leider etwas verwirrend ist (Ich bin aus dem Spec nicht ganz schlau geworden, und auch Google hat nicht viel gebracht), bietet sich an, von vorne herein die Größe der Blocks auf 512 Bytes einzustellen, da dies die einzige Größe ist, die sowohl im Lese- als auch für Schreibbetrieb bei allen Karten garantiert vorhanden ist.

Was mit einer gegebenen Karte genau möglich ist kann man herausfinden, indem man die Kartenregister ausliest, insbesondere das CSD-Register (CSD = Card Specific Data). Hier kann man auch die Kapazität der Karte herausbekommen, aber das ist leider nicht ganz so einfach, da man sich die Größe aus einigen Datenfeldern des Registers zusammenbasteln muss, die nicht mal an Bytegrenzen ausgerichtet sind. Im Beispielcode habe ich die Berechunng der Kartengröße eingebaut, aber für eine genaue Beschreibung sollte man in der SD-Spezifikation nachschlagen.

4.2.1 Blockgröße ändern (CMD16)

CMD16 (SET_BLOCKLEN) ändert die Größe der Datenblocks, die gelesen und geschrieben werden. Welche Blockgrößen unterstützt werden hängt von der Karte ab. Meistens (vielleicht auch immer) ist die Voreinstellung 512 Bytes, aber um sicher zu gehen sollte man diese Größe explizit noch mal festlegen. 512 Bytes ist die einzige Blockgröße, die sowohl im Lese- als auch im Schreibbetrieb von allen Karten unterstützt weden muss.

Der Parameter für das Kommando ist die gewünschte Blockgröße in Bytes, und als Antwort erhält man ein Byte im Format R1.

```
Beispiel:
Send: 0x50 0x00 0x00 0x02 0x00 0x01 (CMD16, Setze Blockgröße auf 512 Bytes)
Recv: 0x00 (R1-Antwort: Alles OK)
```

4.2.2 Einzelnen Block lesen (CMD17)

CMD17 (READ_SINGLE_BLOCK) fordert einen Datenblock von der Karte an. Als Parameter überträgt man die Adresse des Blocks, angegeben in Bytes. Allerdings unterstützen die meisten Karten keine Lesebefehle, die eine physische Blockgrenze auf der Karte überschneiden, daher wird man im Regelfall nur auf Adressen zugreifen die ein Vielfaches von 512 sind.

Die Karte antwortet auf CMD17 wieder mit R1, wenn die Antwort „0x00“ lautet ist alles OK und es geht wie unten beschrieben weiter. Ansonsten ist ein Fehler aufgetreten

und die Transaktion ist beendet, und man sollte versuchen herauszubekommen was schief gelaufen ist.

Die Karte läd jetzt den angeforderten Block in einen internen Zwischenspeicher. Das ist nötig, da bei der Übertragung der Master den Takt vorgibt (und alle SD-Karten bis 25Mhz Taktfrequenz unterstützen müssen), und der Flash-Speicher zu langsam sein könnte um mit der Clockrate mitzuhalten. Sobald die Daten bereit sind, sendet die Karte ein Data Token (0xfe), direkt gefolgt vom Datenblock und zwei CRC-Bytes. Falls ein Fehler aufgetreten ist, z.B. weil man versucht auf einen Block mit zu großer Adresse zuzugreifen oder es einen Lesefehler gab, erhält man statt dessen ein Error Token. Die CRC-Bytes am Ende der Übertragung müssen gelesen werden, auch wenn man sie sofort verwirft.

```
Beispiel:
Send: 0x51 0x00 0x00 0x08 0x00 0x01 (CMD17, Lese Block ab Byte 2048)
Recv: 0x00 (R1-Antwort: Alles OK)
Recv: 0xfe 0x4e 0x65 0x72 0x64 0x21 ..... 0xa7 0x22
```

4.2.3 Einzelnen Block schreiben (CMD24)

Mit CMD24 (WRITE_BLOCK) kann man einen Datenblock auf die Karte schreiben. Der Befehl übernimmt als Parameter die Byte-Adresse, an die der Block geschrieben werden soll. Hier gilt das gleiche wie beim Lesen von Blöcken, die meisten Karten unterstützen keinen Schreibzugriff der eine physische Blockgrenze kreuzt.

Wie immer sollte die Karte möglichst mit 0x00 antworten, falls nicht ist die Transaktion zu Ende und es können keine Daten geschrieben werden. Falls alles OK ist kann man jetzt den Datenblock an die Karte senden. Dabei muss man ein Data Token voranstellen (0xfe), damit die Karte den Beginn des Blocks erkennen kann. Als letztes muss man zwei CRC-Bytes senden, die jedoch von der Karte ignoriert werden wenn man die CRC-Prüfung nicht ausdrücklich eingeschaltet hat.

Sobald das letzte Byte übertragen ist, schickt die Karte ein „Data Response“-Byte. Die ersten drei Bit dieses Bytes sollen ignoriert werden, der Rest gibt Auskunft darüber, ob der Block geschrieben werden kann. Zur Überprüfung dieser Antwort ist es vermutlich am einfachsten, die ersten drei Bits vor dem Auswerten zu löschen, indem das Byte mit 0x1f und-verknüpft wird. Ist das Ergebnis 0x05 wurde der Datenblock akzeptiert, ansonsten gab es einen Fehler.

Nach der Data Response zieht die Karte solange MISO auf Low (sendet Nullbytes), bis der interne Schreibvorgang tatsächlich abgeschlossen ist. In dieser Zeit darf der Karte kein weiteres Kommando geschickt werden. Sobald die Karte wieder bereit ist wird MISO High.

```
Beispiel:
Send: 0x58 0x00 0x00 0x10 0x00 0x01 (CMD24, Schreibe Block ab Byte 4096)
Recv: 0x00 (R1-Antwort: Alles OK)
Send: 0xfe 0x08 0x15 0x47 0x11 ..... 0xde 0xad
Recv: 0xe5 0x00 0x00 0x00 ..... 0x00 0x00 0xff
```

5 Dateisystem?

Jetzt wissen wir, wie man rohe Daten auf die Karte schreibt und davon liest. Aber wenn man z.B. Daten mit einem PC austauscht, will man meistens die Daten auf der Karte in Dateien schreiben und daraus lesen. Die meisten SD-Karten sind dazu mit dem FAT-Dateisystem formatiert. Manche Karten enthalten einen Masterbootsektor mit Partitionstabelle, wie bei einer Festplatte, andere benutzen den gesamten Platz für das Dateisystem, wie bei einer Floppy. Welches Format benutzt wird kann aus dem CSD-Register gelesen werden.

Auch wenn man nur rohe Daten auf der Karte ablegen will, ist es zu empfehlen, dabei nicht die ersten 512 Bytes der Karte zu überschreiben, da manche Kartenleser angeblich Karten ohne gültigen Bootsektor nicht akzeptieren. Außerdem ist das Format der Karte meistens so angepasst, dass Zuordnungseinheiten im Dateisystem mit den physischen Blöcken auf der Karte übereinstimmen, um die Performance zu optimieren. Neuformatieren der Karte könnte deshalb dazu führen, dass die Lese- und Schreibgeschwindigkeit der Karte abnimmt, wenn mit dem Dateisystem gearbeitet wird.

Eine genaue Beschreibung des FAT-Dateisystems würde hier aber zu weit führen. Im Internet kann man viel an Dokumentation und Erklärungen zu diesem Thema finden, und auch kleine Bibliotheken die für den ATMega benutzbar sind. Der Beispelcode zu dieser Ausarbeitung enthält auch eine FAT-Bibliothek, die jedoch nicht sehr flexibel und nicht gut getestet ist.

6 Quellenangaben

- "Physical Layer Simplified Specification Version 2.00", SD Card Association, (http://www.sdcard.org/sd_memorycard/SimplifiedPhysicalLayerSpecification.PDF)
- "How To Use MMC/SDC", (http://elm-chan.org/docs/mmc/mmc_e.html)
- Ulrich Radig's MMC/SD-Projekt, (http://www.ulrichradig.de/home/index.php/avr/mmc-sd)
- ATmega8-Datenblatt, Atmel Corporation, (http://www.atmel.com/dyn/resources/prod_documents/doc2486.pdf)
- Wikipedia (jaja, ist keine Quelle) und weitere diverse Informationsquellen im Internet, v.A. gefunden über Google