retroleum.co.uk

electronics » basic info for reading and writing to MMC cards...

I'll try to keep this article "to the point" and platform agnostic.. I guess most people reading this will want the basics and then go do their own thing..

Right then, MMC cards:

- Cheap flash-memory devices, used for digital cameras etc.
- Supply voltage: 3.3 volts.
- Seven contacts.
- Serial transfer protocols.
- Hot pluggable.
- Normally pre-formatted to the FAT12 or FAT16 standard (FAT32 for cards 2GB and above)

MMC cards contain a microcontroller which allows the card to present itself to whatever its connected to much like a hard drive. IE: the host just sees lots of addressable sectors it can read/write to.

Of the access protocols available, the simplest for hobbiests is "SPI mode" and all the following info is specific to this mode. With SPI selected, communication to the card is done serially via 4 pins, configured as "Clock", "Data in", "Data out" and "Card Select". Data direction is fixed and the host always supplies the clock pulses. Data bits are latched into the card on the rising edge of the clock, and should be read on the falling edge. Data is always transmitted in byte-sized units (MSB first) and the clock can be anything up to 20Mhz. A card's CS line should be held low during all operations aimed at it.

```
Contacts on underside of an MMC card (SPI mode specific)
 ________________
!  _
! !_! 7  - D-out [Data out] (Normally high from card)
! !_! 6  - Ground
! !_! 5  - Clk [Host Clock]
! !_! 4  - 3.3 Volt Supply (Host should be able to supply 100mA)
! !_! 3  - Ground
! !_! 2  - D-in [Data In] (Normally high from host)
! !_! 1  - CS [Card Select] (Active low)
\_________________
```

A protocol of command and data packets is used when reading and writing the card. A command is a set of 6 bytes, in the following format:

```
$0 $1 $2 $3 $4 $5
-----------------
xx yy yy yy yy zz <-- CRC byte
 ! !__________!
 !       !
 !       '---------- 32 bit argument [31:0]
 !
 '----------------- Command code byte
```

The command code byte always has bits 7:6 set to "01" and bits 5:0 hold the command number. The argument is a 32 bit value (contents depend on the command). The CRC byte always has bit 0 set to zero, bits 7:1 hold the command packet's CRC checksum. However, this is disabled in SPI mode unless you specifically switch it back on, so can normally be set to 00h.

Following the sending of a command, there is a delay called the Card Response time (NCR) which can take 1 to 8 bytes depending on the command (during this time the D-out line will be high so FFh will be received by the host). The commands shown below have a one byte gap - ideally you should read bytes until you get a valid card response. When sent, this card response is normally a single byte (called "R1") with its MSB at 0. The rest of the bits have the following meaning when set to 1:

```
"R1" bit:   Signifies:
-----------------------------
  6    -     Parameter error
  5    -     Address error
  4    -     Erase sequence error
  3    -     Command CRC error
  2    -     Illegal command
  1    -     Erase reset
  0    -     In idle state
```

Therefore a response of 00h is normally desired.

Initialization:

Upon power up, MMC cards need to be instructed to change to SPI from their default operating mode. The sequence to do so is as follows:

After power on, wait at least a millisecond and set CS and D-in High.
Send 80 clock pulses.
Set CS low and send a "CMD0" (40h,00h,00h,00h,00h,95h*) to reset the card
(The card checks the CS line when CMD0 is received and goes into SPI mode if OK.)
When CMD0 is accepted, the card enters idle state and so responds with "01h"
Repeatedly send CMD1 (41h,00h,00h,00h,00h,00h) and check the response..
When response is 00h, the card is ready (this can take hundreds of milliseconds)

(* Note the command CRC value is not 00h here as the card wont be in SPI mode yet and actually requires a correct CRC code. "95h" is the "hardwired" value, valid only for CMD0.)

Getting card ID info

Once the card is initialized, the first thing you might want to do is send it commands which tell it to identify itself: CMD9 and CMD10 return data packets of 16 bytes each which contain information such as size of card, maker's name in ascii etc. Some key locations in these data packets are given below. Full details can be found in the MMC spec sheet (see links).

When a command has an associated data packet (like CMD9 and CMD10), a data token will follow the command response, this is then followed by the actual data bytes and finally a 16 bit CRC checksum is sent. The data token for CMD9 and CMD10 (as well as CMD17 and CMD24) is 0FEh.

```
Sequence to read a card's "CSD" bytes (capacity etc)

Send: 49h,00h,00h,00h,00h,00h  - CMD9, no args, null CRC
Read: xx                       - NCR Time
Read: xx                       - Get Command Response (Should be 00h)
Read: until FEh received       - Wait for Data token
Read: yy  * 16                 - Get 16 bytes from CSD
Read: zz                       - Read CRC lo byte
Read: zz                       - Read CRC hi byte
```

Among the useful data in the 16 byte packet is the capacity of the card. Unfortunately its a little cryptic and must be decoded thus:

Byte Locations:

06h,07h,08h : (contents AND 00000011 11111111b 11000000b) >> 6 = "Device size (C_Size)"
09h,0ah : (contents AND 00000011 10000000b) >> 7 = "Device size multiplier (C_Mult)"
05h : (contents AND 00001111b) = Sector size ("Read_BL_Len")

When you have the 12 bit "C_Size", 3 Bit "C_Mult" and 4 bit "Read_BL_Len" you need to follow the formula:

Capacity in bytes = (C_Size+1) * (2 ^ (C_Mult+2)) * (2 ^ Read_BL_Len)

(Note: The computed sector size (2 ^ Read_BL_len) is normally 512 bytes)

```
Sequence to read a card's "CID" bytes (name, serial number etc)

Send: 4ah,00h,00h,00h,00h,00h  - CMD10, no args, null CRC
Read: xx                       - NCR Time
Read: xx                       - Command Response (Should be 00h)
Read: until FEh is received    - Wait for Data token
Read: yy  * 16                 - Get 16 bytes from CID
Read: zz                       - Read CRC lo byte
Read: zz                       - Read CRC hi byte
```

Useful locations in the returned data packet:

03h-08h Manufacturers's name in ascii
0ah-0dh Card's 32 bit serial number

<u>Reading a sector:</u>

The command to read single sector is CMD17. The argument is the BYTE address of the sector you wish to read (so set it at sector number * 512). The CRC is 00h as usual.

Example: Reading from sector 1234h

```
Send: 51h,00h,24h,68h,00h,00h  - CMD17, address, null CRC
Read: xx                       - NCR Time
Read: xx                       - Command Response - should be 00h
Read: until FEh is received    - Wait for Data token (see note 1)
Read: yy  * 512                - Get 512 bytes from sector
Read: zz                       - Read CRC lo byte
Read: zz                       - Read CRC hi byte

Note 1: In a simple implementation, you can simply wait for the
data response "FEh" in a loop with a time-out and report a
general error if it isn't received. What actually happens though
is this: If the card is unable to send the requested data, an
error token will be returned instead of the data token. This
is a single byte with the three MSBs set to zero. Other bits
have the following meaning when set:

Error Token Bit   Meaning:
--------------------------
       4    -     Card Locked
       3    -     CC failed
       2    -     Card ECC failed
       1    -     CC error - card controller failure
       0    -     Error
```

<u>Writing a sector:</u>

The command to write a single sector is CMD24. The argument is the same as for reading a sector and the CRC is 00h as normal. Following the command response, a write delay byte is required (send an FFh), then the data token (FEh) should be sent. Next, the data to fill the sector can be sent with a 2 byte CRC following on (ie: send two zeros). After the last CRC byte is received the card will respond with a data response byte with bits in the format "xxx0sss1" Here "xxx" aren't used, bit 4 is zero, bits 3:1 (sss) hold the status code and bit 0 is a one.

```
Status codes: 010 = Data accepted
            : 101 = Data rejected due to CRC error
            : 110 = Data rejected due to write error
```

After the data response is received, the card will start programming the data it buffered into the card. You must now wait for the busy signal to clear (keep reading bytes and wait for a non-zero byte to be returned) before carrying out further operations. Finally, its advisable to check the card's status bytes to check the sector was actually programmed correctly.

Example: Writing to sector 1234h

```
Send: 58h,00h,24h,68h,00h,00h  - CMD24, address, null CRC
Read: xx                       - NCR Time
Read: xx                       - Command Response - should be 00h
Send: FFh                      - One byte gap
Send: FEh                      - Send Data token
Send: yy  * 512                - Send bytes for sector
Send: zz                       - Send (null) CRC lo byte
Send: zz                       - Send (null) CRC hi byte
Read: vv                       - Read packet response (note 2)
Read: until value is NOT 00h   - Read busy status, wait till done
Send: 4dh,00h,00h,00h,00h,00h  - CMD13 = Send status
Read: xx                       - NCR time
Read: nn                       - Read Status Byte "R1" (note 3)
Read: mm                       - Read Status Byte "R2" (""  "")

Note 2:
Packet Response: If (value AND 1Fh) >> 1 = 02h, packet received OK

Note 3:
Status Byte nn is the same format as "R1"
Status Byte mm ("R2") has the following format (set bits indicate errors):

"R2" bit:   Signifies:
-----------------------------
  7    -     Command args out of range / CSD overwrite attempt
  6    -     Erase parameter is bad
  5    -     Write protected area violation
  4    -     Card ECC failed - internal data correction failed
  3    -     CC error - card controller failure
  2    -     General or unknown error occured
  1    -     Write protect erase skip
  0    -     Card is locked
```

You dont have to read and write just single sectors, there are commands to send groups of them - of course this complicates things a little. More information can be found in the links below.

...

Whilst experimenting I made a simple PC parallel port interface and some DOS software to perform the above 3 commands. The interface drops the 5 volt logic voltages from the parallel port down to something suitable for a 3.3 volt device, and uses a 74HC245 IC as a buffer. There's also a PNP transistor to switch the power to the card on and off. You could probably get by without much of the circuit - the drop-down resistors are the only crucial part. Incidentally, an edge connector from a old-type PC floppy drive lead makes a surprisingly decent connector for experimenting with MMC cards, see **this page** for details (note however, such a connector will not offer hot-pluggable connectivity as all the contacts are the same length.)

The MMC program I made should be run from REAL MODE DOS only, like so:

"mmc -i filename.bin" - read a card's CSD/CID info (32 bytes in all) to a file
"mmc -r:wxyz filename.bin" - read the sector specified (hex "wxyz" - always 4 digits!) to a file
"mmc -w:wxyz filename.bin" - write to the sector specified (hex "wxyz" - always 4 digits!) from a file

As it stands, the program is only suitable for testing/experimentation due to its slow speed and the way it (unnecessarily) power cycles/initializes the card every time it is run. Also, as the desired sector number is given as a 16 bit value, only the first 32MB of any card can be accessed (I only had a 32MB card for testing purposes..)

"MMC.EXE" can be downloaded **here**. The zip file also contains the x86 assembly source code and the schematic for the interface.

Further reading:

- ELM - How to use an MMC
- SanDisk's MMC Manual
- Sean Ellis's MMC to serial adapter
- Captain's PIC to MMC circuit and code
- MMC info (in German - Google's translation tools may help:)