Introduction to the MultiMediaCard

J.-M Friedt, May 29, 2002

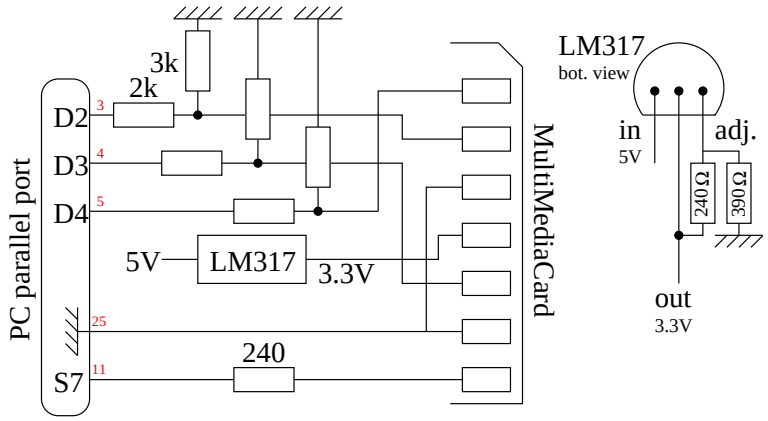
1 Introduction

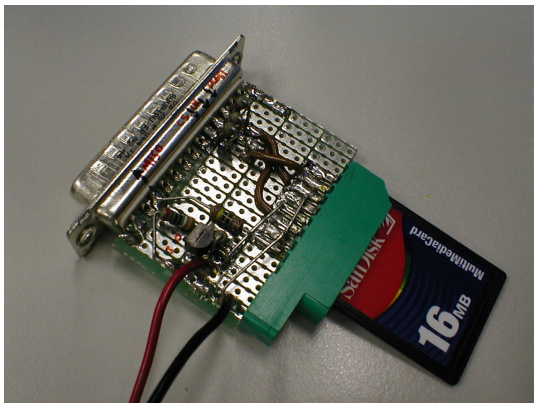
The MultiMediaCard (MMC) is a non-volatile EEPROM based mass storage medium mainly used in mobile phones or digital camera. It features low voltage/low power requirements, very simple connection (7 wires), ease of programming (using Motorola's SPI synchronous protocol) and very large storage capacity (from 16 to 64 MB).

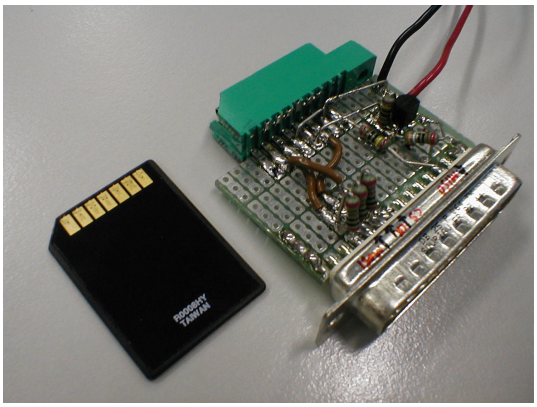
We will first start by using the parallel port of a PC for learning how to access the MMC, reading and writing data to it. The difficulty here mainly resides in shifting the logic levels from the TTL 5 V on the parallel port side to the 3.3 V on the MMC side. We also here develop the basic functions for SPI protocol software emulation since the microcontrollers we are interested in (Motorola 68HC908JB8 and Atmel ATtiny15l) are not equipped with hardware SPI ports.j Such a setup based on the parallel port might be usefull for reading data stored on the MMC by a microcontroller.

2 3.3 V/5 V levels conversion

Although Sandisk advises a quite complex (in terms of number of components) voltage translation circuit based on two transistors (for each of the 4 signals shared between the parallel port and the MMC), we here used a much simpler voltage divider bridge circuit (made of an 2 kΩ resistor connected to signal output and an 3 kΩ resistor connected to ground) for converting the TTL 5 V outputs of the parallel port to 3.3 V, and the internal pull-up of the input port (status port) for keeping its level high expect when the MMC pulls it low (the MMC being protected from the 5 V level by a 240 Ω resistor). The power to the MMC is supplied by the 5 V output of an PC power supply shifted to 3.25 V by an LM317 voltage converter.







3 SPI software emulation

The SPI protocol software emulation on the PC is largely based on the software written by Peter D'Antonio and Daniel Riiff [1]. Their software was adapated to linux and should run (under root) on most PC computers equipped with a parallel port.

The aim of this first program is to store data in the first 512 memory locations (the default MMC block size) and read them in order to check that the data were properly stored.

```
#include "Ports.h"
#include <asm/io.h>

unsigned char InPort(unsigned short port){return inb(port);}
void OutPort(unsigned short port, char val){outb(val,port);}
```

`ports.c` provides the basic I/O ports functions (separated for portability purposes)

```
void OutPort(unsigned short port, char val);

unsigned char InPort(unsigned short port);
```

`ports.h` is the header of the previous functions.

```
#include "SPImaster.h"
#include <stdio.h>
#include <stdlib.h>
#include "Ports.h"
#include "MMC.h"

void RaiseSS(){                                 //SS is pin 5 of LPT1 - output
  char Data;
  Data=InPort(LPT1);Data=(Data|SS);OutPort(LPT1,Data);
}

void LowerSS(){                                  //SS is pin 5 of LPT1 - output
  char Data;
  Data=InPort(LPT1);Data=Data&(~SS);OutPort(LPT1,Data);
}

// SendByte in SPI MASTER fashion - will READ a byte everytime it writes a byte
char SendByte(char byte){
  // CLOCK is on pin 4 of LPT1-output
  // MISO is on pin 11 of LPT1 -input
  // MOSI is on pin 3 of LPT1  -output
  // SS is pin 5 of LPT1 - output
  char input=0,bitin,bit,Data;
  int i;
  Data=InPort(LPT1);
  for(i=0;i<8;i++){                                          //get first bit to send
    bit=(byte & 0x80)>>6;    // checks what MSb is and moves it to 2nd position
    byte = byte << 1;
    Data=Data&~CLOCK;                                                //lower CLOCK
    OutPort(LPT1,Data);
    Data=((Data&~MOSI)|bit);                                 //put DATA on MOSI line
    OutPort(LPT1,Data);
    Data=Data|CLOCK;                                                 //raise CLOCK
    OutPort(LPT1,Data);
    bitin=(((InPort(LPT1+1)^MISO)&MISO)>>7);
    input=input<<1;
    input=input|bitin;
  }
  Data=Data|MOSI;                 //Raise Data - acting like a pull-up resistor
  OutPort(LPT1,Data);
  return input;
}
```

`SPImaster.c` provides the basic SPI functions such as asserting the CS line and reading/writing a byte using the SPI protocol.

```
//FUNCTION PROTOTYPES

char SendByte(char byte);
void RaiseSS();
void LowerSS();

#define LPT1 0x378

#define MOSI 0x02
#define CLOCK 0x04
#define SS 0x08

#define MISO 0x80
```

`SPImaster.h` is the header of the previous functions as well as the definition of SPI protocol signal assignements and parallel port base address.

[1] `http://instruct1.cit.cornell.edu/courses/ee476/FinalProjects/s2000/peterdan/final.htm`

4 Accessing the MMC

All commands sent to the MMC must be 6 bytes long, and are most usually made of a first command byte, followed by an 4-bytes address and terminated by an one byte checksum. In SPI mode, checksum checking is not enabled and the last byte will always be 0xFF, execpt during the first initialization step (when checksum calculation is still enabled).

The answers from the MMC are usually one byte long. The answer is 0xFF when then MMC is busy, so we discard all such responses. The whole communication protocole is finely described in documents available at `http://www.sandisk.com/tech/oem_design/mmc_dc.asp`.

```
/* Implementation of commands needed with the SPI interface of MMC card */

#include <stdio.h>
#include "SPImaster.h"
#include "MMC.h"

int InitMMC(void){                    //raise SS and MOSI for 80 clock cycles:
  int i;                               // SendByte(0xff) 10 times with SS high
  char response=0x01;

  RaiseSS();                              //initialization sequence on PowerUp
  for(i=0;i<=9;i++) SendByte(0xff);
  LowerSS();
  SendCmd(0x00,0,0x95);                //Send Command 0 to put MMC in SPI mode
  if(MMCgetResponse()!=0x01) return false;       //Now wait for READY RESPONSE
  while(response==0x01){
    debug("Snd Cmd1 -- ");                                 // then send CMD1
    RaiseSS();
    SendByte(0xff);
    LowerSS();
    SendCmd(0x01,0x00ffc000,0xff);
    response=MMCgetResponse();
  }
  if(response==0x00) debug("RESPONSE WAS GOOD\n");
  RaiseSS();
  SendByte(0xff);
  debug("MMC INITIALIZED AND SET TO SPI MODE PROPERLY.\n");
  return true;
}

char MMCgetResponse(){                //Response comes 1-8bytes after command
  int i=0;              //the first bit will be a 0, followed by an error code
  char response;                          //data will be 0xff until response
  while(i<=8){
    response=SendByte(0xff);
    if(response==0x00) break;
    if(response==0x01) break;
    i++;
  }
  return response;
}

void SendCmd(char command, int argument, char CRC)          // always MSB first
{char *cmd;
 cmd=&argument;  // break 1 int in 4 bytes: ASSUMES this system uses 4 byte int
 SendByte(command|0x40);
 SendByte(cmd[3]); SendByte(cmd[2]); SendByte(cmd[1]); SendByte(cmd[0]);
 SendByte(CRC);
}

int MMCwriteBlock(char *Block, int address){//WRITE a BLOCK starting at address
  unsigned char busy,dataResp;int count;      //Block size is 512 bytes exactly
  LowerSS();                                              //First Lower SS
  SendCmd(24,address,0xff);                         //Then send write command
  if(MMCgetResponse()==00){            //command was a success - now send data
    SendByte(0xfe);                              //start with DATA TOKEN = 0xFE
    for(count=0;count<BLOCK;count++){SendByte(Block[count]);}   //now send data
    SendByte(0xff);                        //data block sent - now send checksum
    SendByte(0xff);
    do {dataResp=SendByte(0xff);} while (dataResp==0xff); // read DATA RESPONSE
    debug("Write: response: %d\n",(unsigned char)dataResp);
    do {busy=SendByte(0xff);} while(busy==0);  // a 0 indicates the MMC is BUSY
    dataResp=dataResp&0x0f; //mask the high byte of the DATA RESPONSE token
    RaiseSS();
    SendByte(0xff);
    if(dataResp==0x0b)
      {debug("DATA WAS NOT ACCEPTED BY CARD -- CRC ERROR\n");return false;}
    if(dataResp==0x05) {debug("Done writing %d bytes\n",BLOCK);return true;}
    debug("Invalid data Response token: %d.",dataResp);return false;
 }
  debug("Command 24 (Write) was not received by the MMC.\n");
  return false;
}

void MMCreadBlock(int address,char *buffer){ //READ a BLOCK starting at address
  char busy;int count;                      //Block size is 512 bytes exactly
  LowerSS();                                              //First Lower SS
  SendCmd(17,address,0xff);                         //Then send write command
  debug("Now reading from %d\n",address);
  do {busy=SendByte(0xff);} while (busy==-1);
  debug("response read1 (0x00): %d\n",busy&0xff);
  do {busy=SendByte(0xff);} while (busy==-1);
  debug("response read2 (0xfe): %d\n",busy&0xff);
  for(count=0;count<512;count++){                              //now send data
     printf("%d  ",SendByte(0xFF));
    }
  do {busy=SendByte(0xff);} while (busy==-1);
  debug("\nresponse read3 (CRC): %d\n",busy&0xff);
  do {busy=SendByte(0xff);} while (busy==-1);
  debug("response read4 (CRC): %d\n",busy&0xff);
  printf("response read5 %d\n",SendByte(0xff));           // REQUIRED: WHY ???
  debug("Read (Cmd 17) done.\n");
  // return false;
}

void MMCstat()
{char busy;
  LowerSS();                                              //First Lower SS
  SendCmd(13,0,0xff);                               //Then send write command
  do {busy=SendByte(0xff);} while (busy==-1);
  printf("Status1: %d - ",busy);printf("Status2: %d\n",SendByte(0xff));
  printf("Status3: %d\n",SendByte(0xff));                 // REQUIRED: WHY ???
}
```

`MMC.c` provides the basic MMC-related functionalities (writing and reading a bloc from the MMC and initializing SPI mode).

```
/*MMC.h - header file for MMC interface
*/

#define debug printf
#define BLOCK 512

int InitMMC(void);
char MMCgetResponse(void);
void SendCmd(char command, int argument, char CRC);
int MMCwriteBlock(char *Block, int address);
void MMCreadBlock(int address,char *buffer);
void MMCstat();

#define true 1
#define false 0
```

`MMC.h` is the header of the previous functions.

```
/* This program should program the SanDisk MMC -- there is no bounds checking,
   so make sure that the size does not exceed the size of the MMC            */

#include <stdio.h>
#include <stdlib.h>
// #include <fstream>
#include <string.h>
#include <unistd.h>
#include <sys/io.h>// jmfriedt: ioperm
#include <math.h>// jmfriedt: sin()
#include "SPImaster.h"
#include "MMC.h"

int main(void){
  char buffer[BLOCK];              // (default BLOCK size of MMC is 512 bytes)
  int address,i;

 if (sizeof(int)!=4) printf("WARNING: int size must be at least 4\n");
 ioperm(LPT1,3,1);                     // linux specific: enable access to LPT1

 //Initialize MMC so Data can be written
  if(InitMMC()==false){debug("Could not initialize MMC.\n");return(1);}
```

```
for (i=0;i<BLOCK;i++) buffer[i]=(char)(127.*sin(((float)i)/50));
address=0; MMCwriteBlock(buffer,address); MMCstat(); // fill mem0 (with sin.)
           MMCreadBlock(address,buffer);  MMCstat();

for (i=0;i<BLOCK;i++) buffer[i]=BLOCK-i;     // fill mem512 (with lin. func.)
address=512; MMCwriteBlock(buffer,address); MMCstat();
             MMCreadBlock(address,buffer);
```

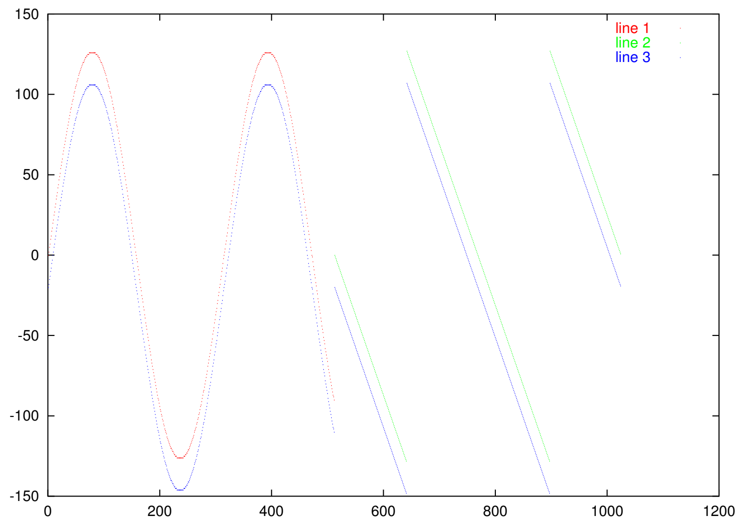
```
  address=0;   MMCreadBlock(address,buffer);
  address=512; MMCreadBlock(address,buffer);

  debug("DONE! \n");
 return(0);
}
```

MMCprog.c

```
all:MMCprog

MMCprog: MMC.o SPImaster.o ports.o MMCprog.c
gcc -Wall -O -o MMCprog MMCprog.c ports.o MMC.o SPImaster.o

MMC.o: MMC.c MMC.h
gcc -Wall -O -c -I. MMC.c
```

```
SPImaster.o: SPImaster.c SPImaster.h
gcc -Wall -O -c -I. SPImaster.c

ports.o: ports.c
gcc -Wall -O -c -I. ports.c

clean:
\rm *.o MMCprog
```

Makefile for compiling the MMCprog application.



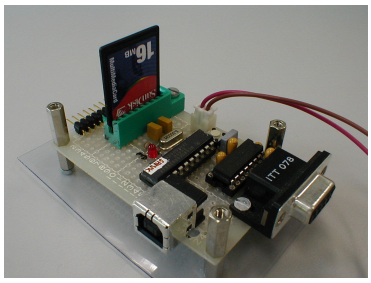
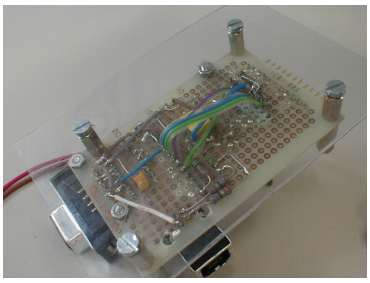
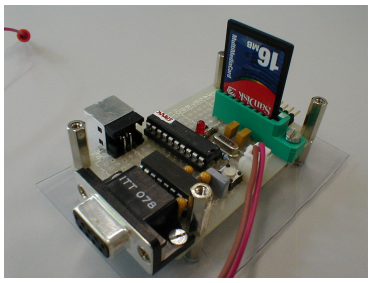
Example of a sine wave stored in the first block (512 bytes) of the MMC, followed by a linear function stored in the second block. Each block is read individually after being written (red and green curves) and then read sequentially (blue line, shifted by 20 units lower) (output of the MMCprog application).

```
Snd Cmd1 -- Snd Cmd1 -- Snd Cmd1 -- Snd Cmd1 -- Snd Cmd1 -- Snd Cmd1 --
Snd Cmd1 -- Snd Cmd1 -- Snd Cmd1 -- Snd Cmd1 -- Snd Cmd1 -- Snd Cmd1 --
Snd Cmd1 -- Snd Cmd1 -- Snd Cmd1 -- Snd Cmd1 -- Snd Cmd1 -- Snd Cmd1 --
Snd Cmd1 -- Snd Cmd1 -- Snd Cmd1 -- Snd Cmd1 -- Snd Cmd1 -- Snd Cmd1 --
Snd Cmd1 -- Snd Cmd1 -- Snd Cmd1 -- Snd Cmd1 -- Snd Cmd1 -- Snd Cmd1 --
Snd Cmd1 -- Snd Cmd1 -- Snd Cmd1 -- Snd Cmd1 -- RESPONSE WAS GOOD
MMC INITIALIZED AND SET TO SPI MODE PROPERLY.
Write: response: 229
Done writing 512 bytes
Status1: 0 - Status2: 0
Status3: -1
Now reading from 0
response read1 (0x00): 0
response read2 (0xfe): 254
[*** data1 ****]
response read3 (CRC): 229
response read4 (CRC): 18
response read5 -1
Read (Cmd 17) done.
Status1: 0 - Status2: 0
Status3: -1
Write: response: 229
Done writing 512 bytes
Status1: 0 - Status2: 0
Status3: -1
```

```
Now reading from 512
response read1 (0x00): 0
response read2 (0xfe): 254
[*** data2 ****]
response read3 (CRC): 26
response read4 (CRC): 140
response read5 -1
Read (Cmd 17) done.
Now reading from 0
response read1 (0x00): 0
response read2 (0xfe): 254
[*** data3a ****]
response read3 (CRC): 229
response read4 (CRC): 18
response read5 -1
Read (Cmd 17) done.
Now reading from 512
response read1 (0x00): 0
response read2 (0xfe): 254
[*** data3b ****]
response read3 (CRC): 26
response read4 (CRC): 140
response read5 -1
Read (Cmd 17) done.
DONE!
```

Output of MMCprog, where the data displayed in the previous graph have been replaced by `[*** DATA?? ***]`.

5 Adapting the MMC to the 68HC08JB8 microcontroller

The 68HC08JB8 microcontroller uses 3.3 V levels on its I/O pins: no level shifting circuitry is thus necessary between the MMC and the microcontroller. The JB8 version of the 68HC08 does not include an SPI port though, so software emulation of this protocol is required as in the case of the parallel port. The only additional hardware required, appart from the MMC card connector (part of an ISA bus connector in our case) is two pull-up resistors (we chose 2 kΩ, should be less than 10 kΩ) since the 3.3 V regulated output of the 68HC08JB8 can be use as a power supply of the MMC.

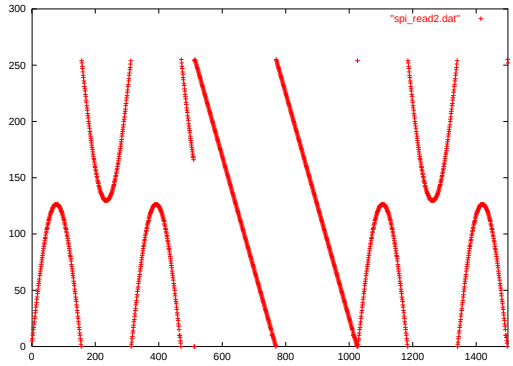
```
; send a byte via RS232 (9600 bps); PTA0: RS232 TxD
; PTA1: Data In (MOSI); PTA2: CS; PTA3: CK; PTA4: Data Out (MISO)

.area code (ABS)
.org 0hdc00
MOSI=0x40 ; value to be sent by SPI
MISO=0x41 ; value read by SPI
ADDR4=0x44 ; MSB of memory address
ADDR3=0x43 ; memory address
ADDR2=0x42 ; memory address (MUST be multiple of 2 for 512 bytes blocks)

start: ldhx  #0x0140 ; TXS : (SP)<-(H:X)-1 => STACK=0x013f
txs ; reset stack pointer
; mov   #0x01,CONFIG1 ; disable COP watchdog, CONFIG1=0x001f

mov   #0x00,0x0003 ; PTD0/1 lo => LED lit
mov   #0x03,0x0007 ; DDRD: PTD0/1 as output

mov   #0x01,0xff ; PTA: PTA0 hi
mov   #0x0f,0x0004 ; DDRA: PTA0 ,1 ,2, 3 as output

bsr mmc_ini
mov #0x00,ADDR4 ; address 0
mov #0x00,ADDR3 ; address 0
loop: mov #0x00,ADDR2 ; address 0
jsr mmc_read
mov #0x02,ADDR2 ; address 512=2*256  ASSUMES A BLOCK OF 512 B
jsr mmc_read
bra   loop

; MMC intialization
mmc_ini:lda #80 ; 80 clock oscillations with CS and MOSI hi
bset #0x01,*0x00 ; MOSI -> hi
bset #0x02,*0x00 ; CS# -> hi
init_ck:bclr #0x03,*0x00 ; CK -> lo
nop
bset #0x03,*0x00 ; CK -> hi
deca
bne init_ck
bclr #0x02,*0x00 ; CS# -> lo
cmd0: mov  #0x40,MOSI ; value to be sent (cmd0 | 0x40)
bsr spi_snd
mov  #0x00,MOSI ; value to be sent (@0)
bsr spi_snd
mov  #0x00,MOSI ; value to be sent (@1)
bsr spi_snd
mov  #0x00,MOSI ; value to be sent (@2)
bsr spi_snd
mov  #0x00,MOSI ; value to be sent (@3)
bsr spi_snd
mov  #0x95,MOSI ; value to be sent (CRC)
bsr spi_snd
bset #0x01,*0x03 ; switch LED off
resp0: mov  #0xff,MOSI ; read answer (should be 01)
bsr spi_snd
lda MISO
cmp #0xff
beq resp0
jsr send ; send on RS232 port the answer (should be 1)
; from now on we send command 1
cmd1: bset #0x02,*0x00 ; CS=hi
mov  #0xff,MOSI ; value to be sent
bsr spi_snd
bclr #0x02,*0x00 ; CS=lo
mov  #0x41,MOSI ; value to be sent (cmd1 | 0x40)
bsr spi_snd
mov  #0x00,MOSI ; value to be sent (@0)
bsr spi_snd
mov  #0xff,MOSI ; value to be sent (@1)
bsr spi_snd
mov  #0xc0,MOSI ; value to be sent (@2)
bsr spi_snd
mov  #0x00,MOSI ; value to be sent (@3)
bsr spi_snd
mov  #0xff,MOSI ; value to be sent (CRC)
bsr spi_snd
resp1: mov  #0xff,MOSI ; read answer (should be 00)
bsr spi_snd
lda MISO
cmp #0xff ; wait for MMC to be ready ... (retry read)
beq resp1
cmp #0x01
beq cmd1 ; keep on sending CMD1 until answer is NOT 1
jsr   send ; send value to serial port (should be 0)
bclr #0x01,*0x03 ; switch LED on
bset #0x02,*0x00 ; CS=hi
mov  #0xff,MOSI ; send another 8 clock pulses
bsr spi_snd
rts

; sends byte stored in RAM location 0x40 and puts in 0x41 the byte read
spi_snd:
lda #0x08 ; 8 bits in 1 bytes
nxt_spi:bclr #0x03,*0x00 ; CK -> lo
rol MOSI ; send MSB first => roL (NOT roR)
bcc to0  ; branch if 0
bset #0x01,*0x00 ; MOSI=1
bra mosiok
to0: bclr #0x01,*0x00 ; MOSI=0
mosiok: bset #0x03,*0x00 ; CK -> hi
rol MISO ; we receive MSB first => 8x roL
brclr  #0x04,*0x00,not1; MISO=PA4
bset #0x00,*MISO
bra end1
not1: bclr #0x00,*MISO ; just in case C=1 before roL
end1: deca ; loop n times (C<-dat[A+1] => loop until A=0)
bne nxt_spi
bset #0x01,*0x00 ; MOSI -> hi
rts

mmc_read:bclr #0x02,*0x00 ; CS=lo
mov  #0x51,MOSI ; value to be sent (cmd17 | 0x40) (0d17=0x11)
bsr spi_snd
lda  ADDR4
sta MOSI  ; => value to be sent (@0)
bsr spi_snd
lda  ADDR3
sta MOSI  ; => value to be sent (@1)
bsr spi_snd
lda  ADDR2
sta MOSI  ; => value to be sent (@2)
bsr spi_snd
mov  #0x00,MOSI ; value to be sent (@3)
bsr spi_snd
mov  #0xff,MOSI ; value to be sent (CRC)
bsr spi_snd
resp17a:mov  #0xff,MOSI ; read answer (should be 00)
bsr spi_snd
lda MISO
cmp #0xff ; answers 0x04 if we sent a wrong cmd code (for
beq resp17a ; example 0x57 instead of 0x51) and 0x20 if the
bsr  send ; requested @ is not valid (1 instead of 0)
resp17b:mov  #0xff,MOSI ; read answer (should be fe)
bsr spi_snd
lda MISO
cmp #0xff
beq resp17b
bsr  send
```

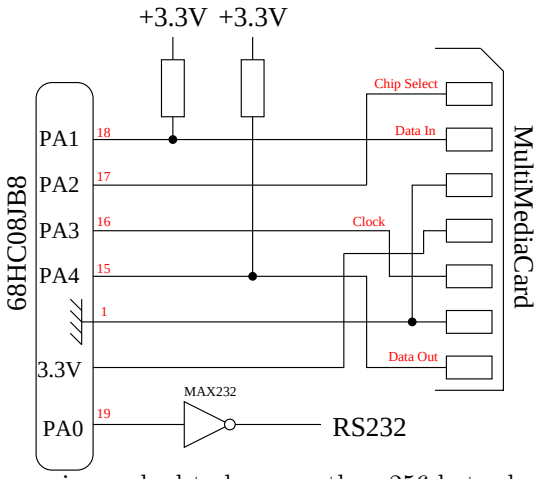
```
ldhx #0 ; 512 bytes in 1 block (default block size)
resp17c:mov  #0xff,MOSI ; read value
bsr spi_snd
lda MISO
bsr  send
aix #0x01
cphx #512 ; ASSUMES A BLOCK OF 512 BYTES
bne resp17c
bset #0x02,*0x00 ; CS=hi
rts

; send a byte via RS232 (9600 bps)
send: pshx
ldx #0x08 ;   snd through PTA0 the content of Acc (@9600)
bclr #0x00,*0x00 ;   PTA: PTA0 lo : START bit
looprs: bsr delay ; X
bsr delay ; X
rora ; 1 rotate right Acc through carry
bcc bit0 ; 3 branch if carry is clear (is A&1=0)
```

```
bset #0x00,*0x00
bra bit1      ; 3
bit0: bclr  #0x00,*0x00 ; 4 PTA0=lo
bit1: dbnzx looprs ; 3   --> sum=11 or 14
fin: bsr delay
bsr delay
bset   #0x00,*0x00 ; PTA: PTA0 hi : STOP bit
bsr delay
bsr delay
pulx
rts

delay: pshx ; 2+4 for bsr   (12+2*((X*9)+15))*.3333=833
ldx #0h0f ; 3                                        104
loopx: nsa ; 3    => Xinit=0x88 for 1200
nsa ; 3            =0x0f for 9600
dbnzx loopx ; 3
pulx ; 2
rts ; 4
```

`spi_read.asm`: 68HC08 assembly program for reading data from an MMC, including SPI and RS232 (9600 bauds, N81) software emulations.

The following diagrams display on the right the connexion of the MMC to the 68HC908JB8 microcontroller (quartz, USB and reset logic not shown, cf document *Introduction to the Motorola 68HC908JB8* for a full description of the hardware around the 68HC908JB8 microcontroller), and on the left the data read from the MMC after executing `spi_read` while the MMC with data stored using the parallel port (cf chapter 4) was connected to the microcontroller.





Since the program has become too long and some jumps had to be more than 256 bytes long, some `jsr` instructions had to be used instead of `bsr` and the location of the program in EEPROM using the `.org 0hdc00` header is mandatory.

I send back to RS232 all meaningful responses from the MMC: the MMC should first answer "1" to command CMD1, then "0" to confirm it switched to SPI mode. The read instruction provides first a "0" answer followed by "FE". Any other answer means an error occured (for example "4" means an illegal instruction was sent, such as sending 57 instead of 51 as the first byte for the read instructions (CMD17 in decimal, ie 11 in hexadecimal), while "20" means an illegal block address was provided, such as asking for memory location 1 instead of 0).

Once we are able to read from the MMC, we also want to be able to write to it.

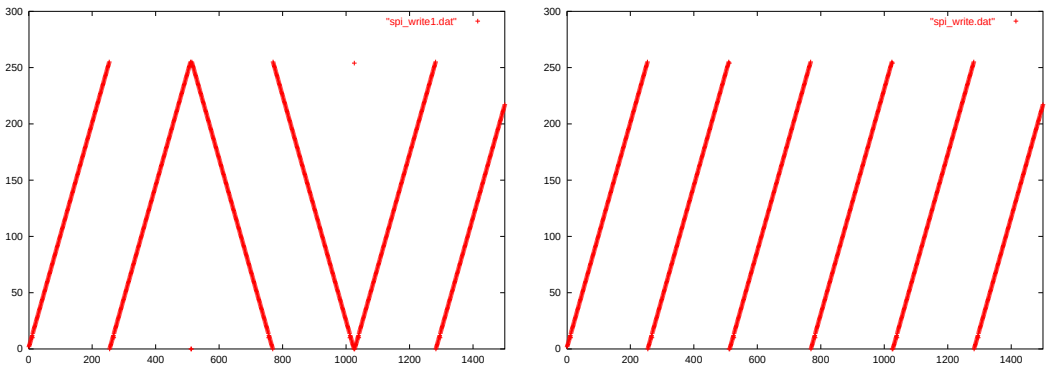
```
; send a byte via RS232 (9600 bps); PTA0: RS232 TxD
; PTA1: Data In (MOSI); PTA2: CS; PTA3: CK; PTA4: Data Out (MISO)

.area code (ABS)
.org 0hdc00
MOSI=0x40 ; value to be sent by SPI
MISO=0x41 ; value read by SPI
ADDR4=0x44 ; MSB of memory address
ADDR3=0x43 ; memory address
ADDR2=0x42 ; memory address (MUST be multiple of 2 for 512 bytes blocks)

start: ldhx  #0x0140 ; TXS : (SP)<-(H:X)-1 => STACK=0x013f
txs ; reset stack pointer
; mov   #0x01,CONFIG1 ; disable COP watchdog, CONFIG1=0x001f

mov   #0x00,0x0003 ; PTD0/1 lo => LED lit
mov   #0x03,0x0007 ; DDRD: PTD0/1 as output

mov   #0x01,0xff ; PTA: PTA0 hi
```

```
mov   #0x0f,0x0004 ; DDRA: PTA0 ,1 ,2, 3 as output

bsr mmc_ini
mov #0x00,ADDR4 ; address 0
mov #0x00,ADDR3 ; address 0
mov #0x00,ADDR2 ; address 0
jsr mmc_writ
mov #0x02,ADDR2 ; address 512=2*256  ASSUMES A BLOCK OF 512 B
jsr mmc_writ
bset #0x01,*0x03 ; switch LED off
loop: bra   loop

; MMC intialization
mmc_ini:lda #80 ; 80 clock oscillations with CS and MOSI hi
bset #0x01,*0x00 ; MOSI -> hi
bset #0x02,*0x00 ; CS# -> hi
init_ck:bclr #0x03,*0x00 ; CK -> lo
nop
bset #0x03,*0x00 ; CK -> hi
```

```
deca
bne init_ck
bclr #0x02,*0x00 ; CS# -> lo
cmd0: mov  #0x40,MOSI ; value to be sent (cmd0 | 0x40)
bsr spi_snd
mov  #0x00,MOSI ; value to be sent (@0)
bsr spi_snd
mov  #0x00,MOSI ; value to be sent (@1)
bsr spi_snd
mov  #0x00,MOSI ; value to be sent (@2)
bsr spi_snd
mov  #0x00,MOSI ; value to be sent (@3)
bsr spi_snd
mov  #0x95,MOSI ; value to be sent (CRC)
bsr spi_snd
bset #0x01,*0x03 ; switch LED off
resp0: mov  #0xff,MOSI ; read answer (should be 01)
bsr spi_snd
lda MISO
cmp #0xff
beq resp0
jsr send ; send on RS232 port the answer (should be 1)
; from now on we send command 1
cmd1: bset #0x02,*0x00 ; CS=hi
mov  #0xff,MOSI ; value to be sent
bsr spi_snd
bclr #0x02,*0x00 ; CS=lo
mov  #0x41,MOSI ; value to be sent (cmd1 | 0x40)
bsr spi_snd
mov  #0x00,MOSI ; value to be sent (@0)
bsr spi_snd
mov  #0xff,MOSI ; value to be sent (@1)
bsr spi_snd
mov  #0xc0,MOSI ; value to be sent (@2)
bsr spi_snd
mov  #0x00,MOSI ; value to be sent (@3)
bsr spi_snd
mov  #0xff,MOSI ; value to be sent (CRC)
bsr spi_snd
resp1: mov  #0xff,MOSI ; read answer (should be 00)
bsr spi_snd
lda MISO
cmp #0xff ; wait for MMC to be ready ... (retry read)
beq resp1
cmp #0x01
beq cmd1 ; keep on sending CMD1 until answer is NOT 1
jsr   send ; send value to serial port (should be 0)
bclr #0x01,*0x03 ; switch LED on
bset #0x02,*0x00 ; CS=hi
mov  #0xff,MOSI ; send another 8 clock pulses
bsr spi_snd
rts

; sends byte stored in RAM location 0x40 and puts in 0x41 the byte read
spi_snd:
lda #0x08 ; 8 bits in 1 bytes
nxt_spi:bclr #0x03,*0x00 ; CK -> lo
rol MOSI ; send MSB first => roL (NOT roR)
bcc to0  ; branch if 0
bset #0x01,*0x00 ; MOSI=1
bra mosiok
to0: bclr #0x01,*0x00 ; MOSI=0
mosiok: bset #0x03,*0x00 ; CK -> hi
rol MISO ; we receive MSB first => 8x roL
brclr  #0x04,*0x00,not1; MISO=PA4
bset #0x00,*MISO
bra end1
not1: bclr #0x00,*MISO ; just in case C=1 before roL
end1: deca ; loop n times (C<-dat[A+1] => loop until A=0)
bne nxt_spi
bset #0x01,*0x00 ; MOSI -> hi
rts

mmc_writ:bclr #0x02,*0x00 ; CS=lo
mov  #0x58,MOSI ; value to be sent (cmd24 | 0x40) (0d24=0x18)
bsr spi_snd
lda  ADDR4
sta MOSI  ; => value to be sent (@0)
bsr spi_snd
lda  ADDR3
sta MOSI  ; => value to be sent (@1)
bsr spi_snd
lda  ADDR2
sta MOSI  ; => value to be sent (@2)
bsr spi_snd
mov  #0x00,MOSI ; value to be sent (@3)
bsr spi_snd
mov  #0xff,MOSI ; value to be sent (CRC)
bsr spi_snd
resp17a:mov  #0xff,MOSI ; read answer (should be 00)
bsr spi_snd
lda MISO
cmp #0xff ; answers 0x04 if we sent a wrong cmd code (for
beq resp17a ; example 0x57 instead of 0x51) and 0x20 if the
bsr  send ; requested @ is not valid (1 instead of 0)
mov  #0xfe,MOSI ; send $fe (block start)
bsr spi_snd
ldhx #0 ; 512 bytes in 1 block (default block size)
resp17b:stx  MOSI ; value to be written
bsr spi_snd
aix #0x01
cphx #512 ; ASSUMES A BLOCK OF 512 BYTES
bne resp17b
mov  #0xff,MOSI ; send CRC1
bsr spi_snd
mov  #0xff,MOSI ; send CRC2
bsr spi_snd
resp17c:mov  #0xff,MOSI ; read answer (might be 229 ? => the low nibble
bsr spi_snd ; of this answer MUST be 5 => data accepted)
lda MISO
cmp #0xff ; answer
beq resp17c
bsr  send
resp17d:mov  #0xff,MOSI ; read answer: repeat while it is 0 (busy)
jsr spi_snd
lda MISO
cmp #0x0 ; answer: if 0 => MMC is busy saving data
beq resp17d
bset #0x02,*0x00 ; CS=hi
mov  #0xff,MOSI ; send 8 clock pulses
jsr spi_snd
rts

; send a byte via RS232 (9600 bps)
send: pshx
ldx #0x08 ;   snd through PTA0 the content of Acc (@9600)
bclr #0x00,*0x00 ;   PTA: PTA0 lo : START bit
looprs: bsr delay ; X
bsr delay ; X
rora ; 1 rotate right Acc through carry
bcc bit0 ; 3 branch if carry is clear (is A&1=0)
bset #0x00,*0x00
bra bit1      ; 3
bit0: bclr  #0x00,*0x00 ; 4 PTA0=lo
bit1: dbnzx looprs ; 3   --> sum=11 or 14
fin: bsr delay
bsr delay
bset   #0x00,*0x00 ; PTA: PTA0 hi : STOP bit
bsr delay
bsr delay
pulx
rts

delay: pshx ; 2+4 for bsr   (12+2*((X*9)+15))*.3333=833
ldx #0h0f ; 3                                          104
loopx: nsa ; 3    => Xinit=0x88 for 1200
nsa ; 3            =0x0f for 9600
dbnzx loopx ; 3
pulx ; 2
rts ; 4
```

`spi_write.asm`: 68HC08 assembly program for writing data to an MMC, including SPI and RS232 (9600 bauds, N81) software emulations.

Since the 68HC08JB8 does not have 512 bytes of RAM, the only meaningfull data we could store in the MMC was a running counter. We indeed see, after running `spi_write`, that data were written in the MMC. We also see in the log containing the data transmitted by `spi_write` that the MMC asnwered `$e5`, whose lowest nibble is "5" meaning the data were correctly written to the MMC (in the first two banks). One could eventually change the block size using command 16 in order to first fill the 68HC908JB8 RAM and once the necessary number of values have been stored in RAM, dump them to the MMC. Otherwise since the MMC communicates using a synchronous protocol with an unrestricted delay between clock cycles, one could simply insert the data acquisition function in the `mmc_write` function and write to the MMC while data are received.



Left: data read from the MMC after executing an older version of the `spi_write` program in which the additional functions required after writing data were not included: the MMC was thus in an unstable state and only the first data block was modified. Right figure: after completing the software with the correct post-writing functions, the data in the first two blocks were updated as expected. The log file (not shown here) obtained by listening to the RS232 port while `spi_write` was executed indeed shows that the MMC answered `$e5` after both blocks were written, indicating that the writing procedure ended correctly.

6 Further projects

- understand OS layout at PC boot time and load OS from MMC.
- port the software to the ATtiny15L (8 pins available, 4 used for SPI protocol and 2 for power supplies $\Rightarrow$ we can have two A/D available if the reset pin can also be used for one of the SPI signals thanks to the pull-up resistors).