Fujitsu Microelectronics Europe
Application Note

MCU-AN-300245-E-V11

F²MC-16FX FAMILY

16-BIT MICROCONTROLLER

ALL SERIES

SPI COMMUNICATION WITH SD-CARD

APPLICATION NOTE

FUJITSU

Revision History

Date	Issue
2009-01-20	V1.0, MSc, First Version
2009-01-26	V1.1, MSc, sw example without command-line

This document contains 35 pages.

Warranty and Disclaimer

To the maximum extent permitted by applicable law, Fujitsu Microelectronics Europe GmbH restricts its warranties and its liability for **all products delivered free of charge** (eg. software include or header files, application examples, target boards, evaluation boards, engineering samples of IC's etc.), its performance and any consequential damages, on the use of the Product in accordance with (i) the terms of the License Agreement and the Sale and Purchase Agreement under which agreements the Product has been delivered, (ii) the technical descriptions and (iii) all accompanying written materials. In addition, to the maximum extent permitted by applicable law, Fujitsu Microelectronics Europe GmbH disclaims all warranties and liabilities for the performance of the Product and any consequential damages in cases of unauthorised decompiling and/or reverse engineering and/or disassembling. **Note, all these products are intended and must only be used in an evaluation laboratory environment**.

1. Fujitsu Microelectronics Europe GmbH warrants that the Product will perform substantially in accordance with the accompanying written materials for a period of 90 days form the date of receipt by the customer. Concerning the hardware components of the Product, Fujitsu Microelectronics Europe GmbH warrants that the Product will be free from defects in material and workmanship under use and service as specified in the accompanying written materials for a duration of 1 year from the date of receipt by the customer.

2. Should a Product turn out to be defect, Fujitsu Microelectronics Europe GmbH´s entire liability and the customer´s exclusive remedy shall be, at Fujitsu Microelectronics Europe GmbH´s sole discretion, either return of the purchase price and the license fee, or replacement of the Product or parts thereof, if the Product is returned to Fujitsu Microelectronics Europe GmbH in original packing and without further defects resulting from the customer´s use or the transport. However, this warranty is excluded if the defect has resulted from an accident not attributable to Fujitsu Microelectronics Europe GmbH, or abuse or misapplication attributable to the customer or any other third party not relating to Fujitsu Microelectronics Europe GmbH.

3. To the maximum extent permitted by applicable law Fujitsu Microelectronics Europe GmbH disclaims all other warranties, whether expressed or implied, in particular, but not limited to, warranties of merchantability and fitness for a particular purpose for which the Product is not designated.

4. To the maximum extent permitted by applicable law, Fujitsu Microelectronics Europe GmbH´s and its suppliers´ liability is restricted to intention and gross negligence.

NO LIABILITY FOR CONSEQUENTIAL DAMAGES

To the maximum extent permitted by applicable law, in no event shall Fujitsu Microelectronics Europe GmbH and its suppliers be liable for any damages whatsoever (including but without limitation, consequential and/or indirect damages for personal injury, assets of substantial value, loss of profits, interruption of business operation, loss of information, or any other monetary or pecuniary loss) arising from the use of the Product.

Should one of the above stipulations be or become invalid and/or unenforceable, the remaining stipulations shall stay in full effect

Contents

1 Introduction

INTRODUCTION

This Application Note describes how to connect MMC/SD cards to a 16FX microcontroller via SPI and gives an idea how to use the driver from application software. A simple command-line user interface allows exploring folders and files, reading, writing, append and copy files, renaming and deleting files or folders, creating folders and formatting the FAT file system. It is also possible to add own commands in the command line structure, so the user has the possibility to test the full function of the MMC/SD implementation in a full functional embedded filesystem.

Supported file systems are FAT, FAT16 and FAT32.

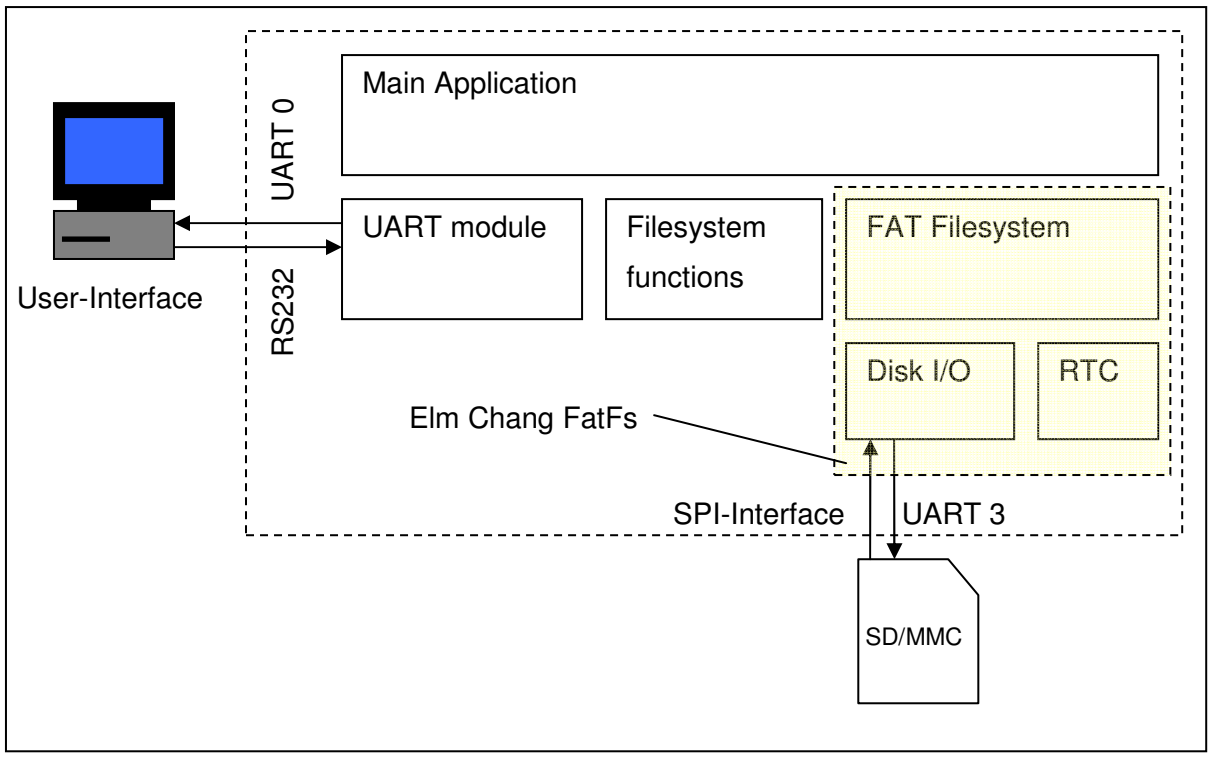
The software is developed for the Fujitsu MB96300 family microcontroller. This example is designed primarily for Fujitsu MB96F348 (SK-16FX-EUROSCOPE Starterkit) and the MB96F338 (SK-144PMC-USB Starterkit) microcontroller.

The FAT file system used within this application note can be downloaded from the developer's website: http://elm-chan.org/fsw/ff/00index_e.html

1.1 Overview

The project is divided in a hardware section and a software section. The hardware section shows some sample connections of the MMC/SD card for two starter kits, explains the SD-card protocol and describes hardware specific configurations in the software part.

The software section explains the implementation of the different software modules which are necessary to use card storage with a FAT filesystem. Most modules are based on the FAT Filesystem (also FatFs called). Main Software Modules are shown below:



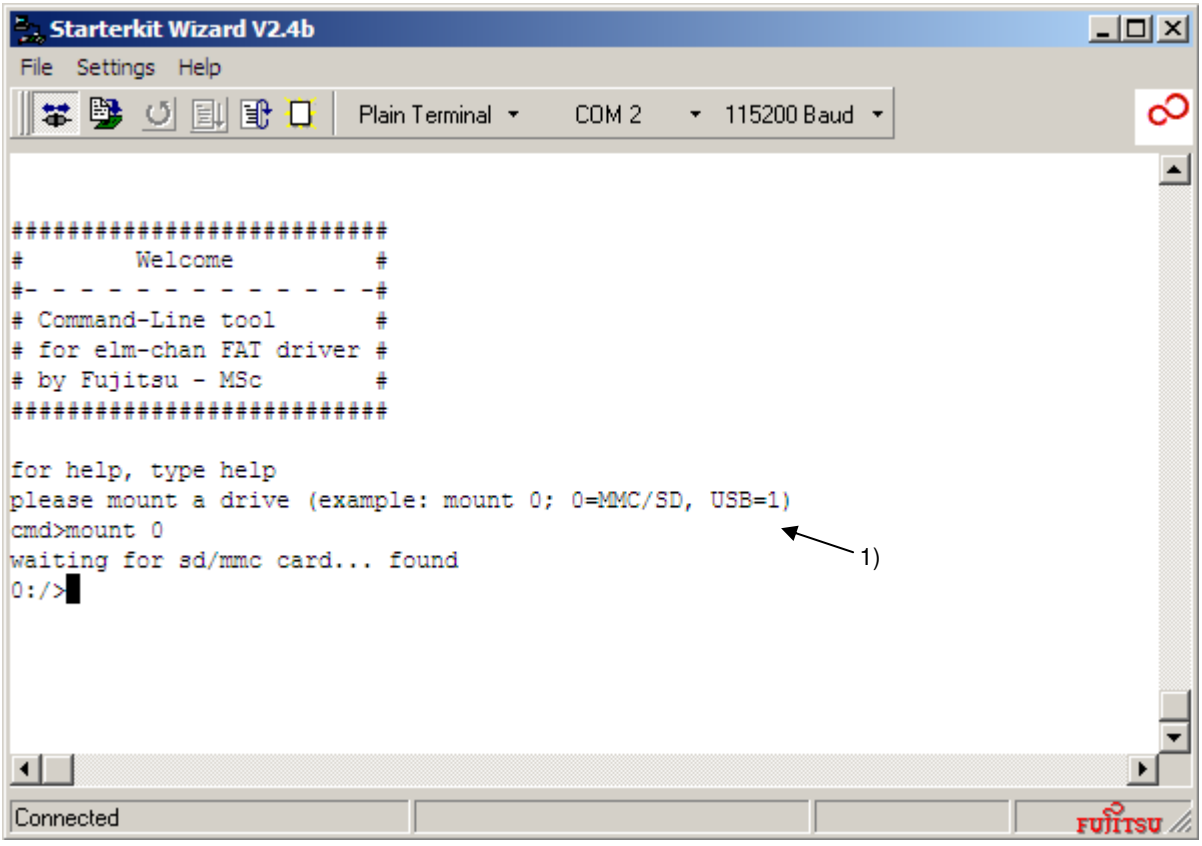
1.2 Quickstart

First the MMC/SD Card connection has to be realized. This can be done by the sample connections in chapter “2 Hardware” for starterkit SK-16FX-EUROSCOPE or SK-16FX-144PMC-USB.

1.2.1 First Start

The application interface is realized as a command line over UART 0 (115200,8N1). Connect a serial cable (RS232) to the PC and use a terminal application like *Starterkit Wizard* to communicate with the MCU. After resetting the MCU a welcome screen should be displayed.

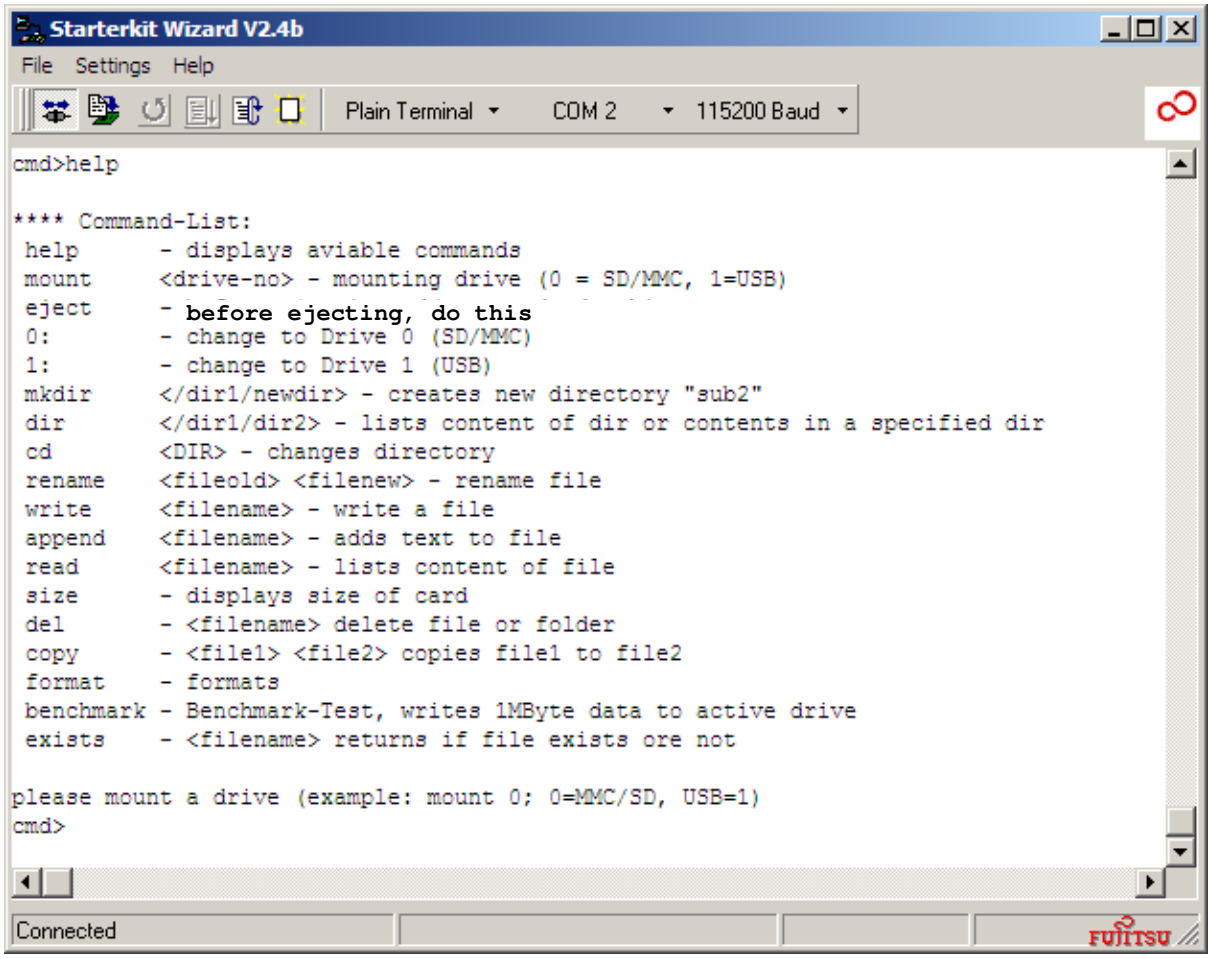
A drive has to be mounted: Insert a MMC/SD card and type in `mount 0`. If the command line displays `0:/>` the card was successfully mounted.



[1] In the terminal application it can be seen that there is the possibility to mount USB. This is only available with 16FX USB devices (MB96330 series) and won’t be work with the MB96340 series.

1.2.2 Available Commands

To display a list of available commands use the command `help`. Help displays all commands and which parameters are required to use them. Some commands can only be done if a drive was mounted. The commands which are working without mounting any drive are `help`, `mount`, `0:` and `1:`.



1.2.3 Syntax

The syntax of the command-line is simple structured. A command is followed with parameters. Command and parameter are divided by one space character.

Command-Line:

```
    No disk is mounted, command-mode only without filesystem features
       cmd>COMMAND PARAMETER1 PARAMETER2 ... PARAMETERn

    Disk was mounted:
       0:/>COMMAND PARAMETER1 PARAMETER2 ... PARAMETERn
```

File structure notion:

```
    0:/mydir/myfile.txt
    |    |      |
    |    |      +--- Filename, has to be fileformat DOS 8.3
    |    |
    |    +---------- Directory in rootdirectory
    |
    +--------------- Drive Number:  0: SD/MMC
                                    1: USB
```

1.2.4 Example: Write a file without command-line

If there is no need for command-line tools, there is only the need of the io-driver, the io-disk-layer and the filesystem to use SD/MMC support. The following files will be needed:

- *mmc.c, mmc.h* SD/MMC io-driver
- *ff.c, ff.c, diskio.c, diskio.h, integer.h* Elm Chan's filesystem (io-layer and filesystem)
- *vectors.c, hw_support.c* Port definitions, timer interrupt, etc.

An example to write a file could be like the following code example:

```
// examples see also Elm Chan's FatFs: http://elm-chan.org/fsw/ff/00index_e.html
#include "ff.h"
#include "string.h"


int writeFile( void )
{
  UINT wlen,bw;
  FIL  myfile;
  FATFS filesystem;
  char mystring[] = "Hello World!";


  WaitMMCSDInit(0); // wait for SD/MMC ready and initialize it

  f_mount(0,&filesystem); //mount filesystem
  // see also: http://elm-chan.org/fsw/ff/en/mount.html

  f_open(&myfile,"0:/text.txt", FA_CREATE_ALWAYS| FA_WRITE); // open file for write
  // see also http://elm-chan.org/fsw/ff/en/open.html

  wlen = strLen(mystring);

  f_write(&myfile,"mytext",wlen,&bw);
  if (bw != wlen) return -1;      // error bytes written not bytes to write
  // see also http://elm-chan.org/fsw/ff/en/write.html

  f_close(&myfile); // close file
  // see also: http://elm-chan.org/fsw/ff/en/close.html

  f_mount(0,NULL); //unmount filesystem


  return 0;
}
```

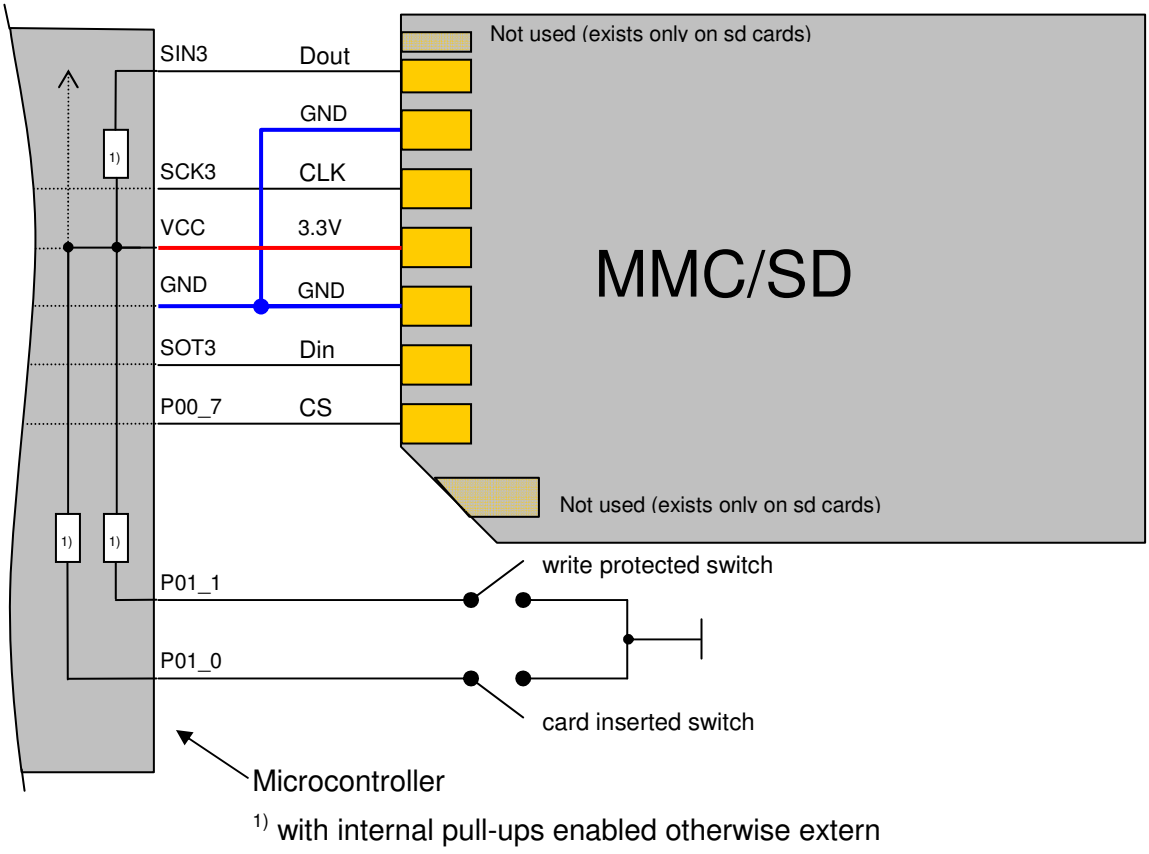
2 Hardware

DESCRIPTION OF MMC/SD CARD HARDWARE IMPLEMENTATION

2.1 Hardware Connection – MMC/SD Interface

The communication between the MCU and MMC/SD card is realized with one internal SPI interface (UART3). For card detection and card write protection there are two switches in the card socket. Take care MMC/SD Cards can only be used in 3,3V systems.

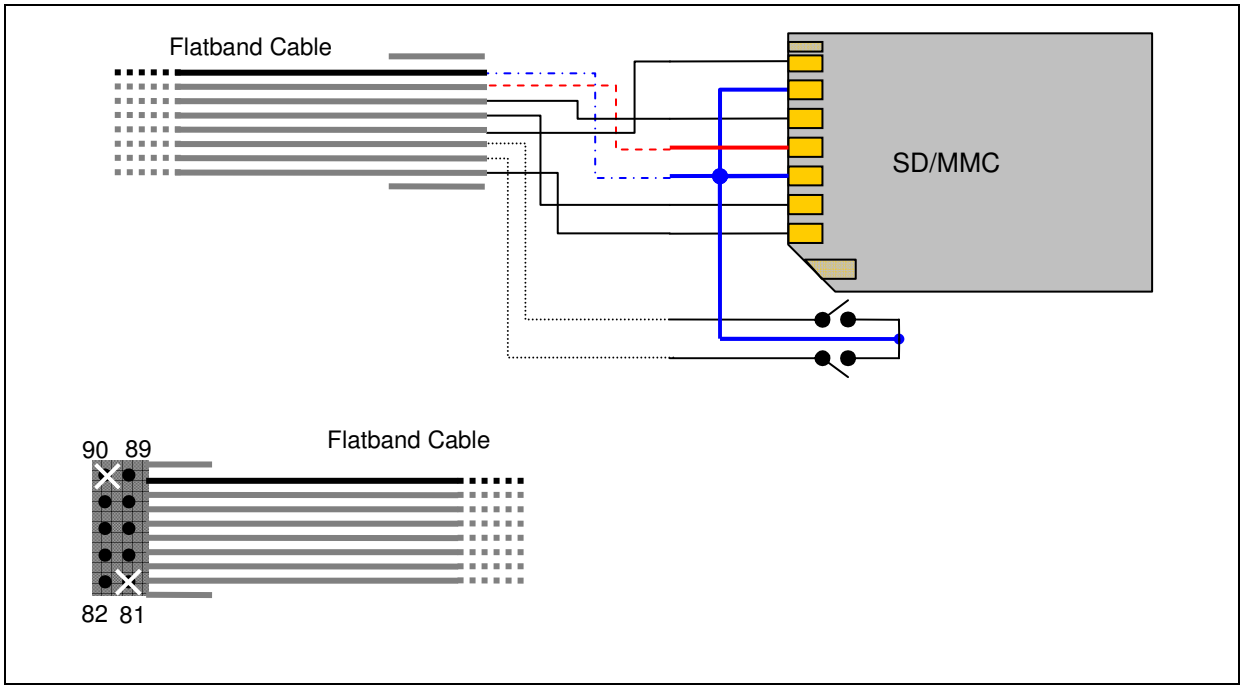
Diagram: How to connect the MMC/SD card (MB96340 Series)



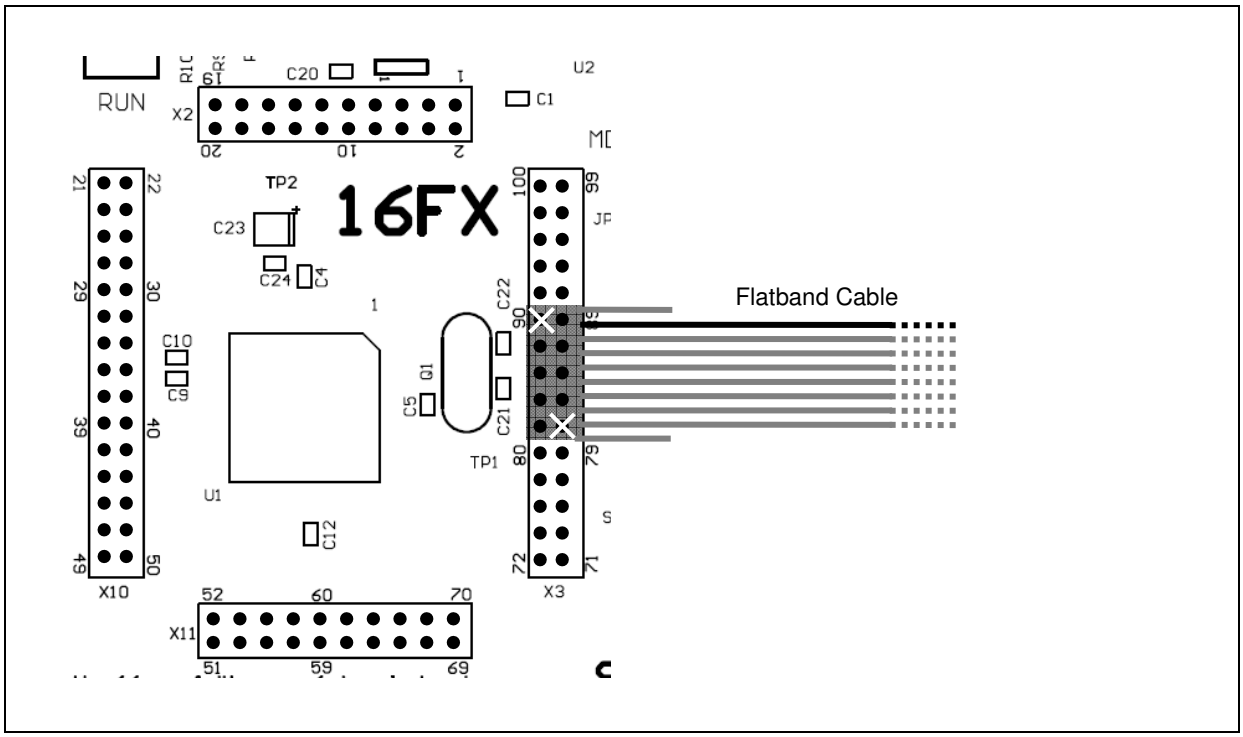
WARNING:	MCU-VCC and SD/MCC card VCC has to be 3.3V otherwise MMC/SD card and MCU causes damage!

2.2 Sample Connection for SK-16FX-EUROSCOPE (MB96340 Series)

To connect a MMC/SD Card to the Starterkit, UART 3 can be used, because next by are also VCC and GND pins. **Do not forget to switch the MCUVCC voltage to 3,3V via jumper JP10**. Connect a MMC/SD Card slot with a flatband cable as it is shown below. On the other side a normal pin header connector can be used. For a stable connection the flatband cable has to be not longer then 20cm.



At the Starterkit use connection header X3. Connect the adapter cable from pin 81 to 90, pin 81 and 90 are not used in this example:



For connections details have a look in the following table. It shows pin-number at the MCU, function of the pin, which port is used and which pin is connected to the MMC/SD Card. Port 0 is also used for the right seven segment display. Pin *P00_P7* is the dot in the display and shows cable select activity of the MMC/SD card.

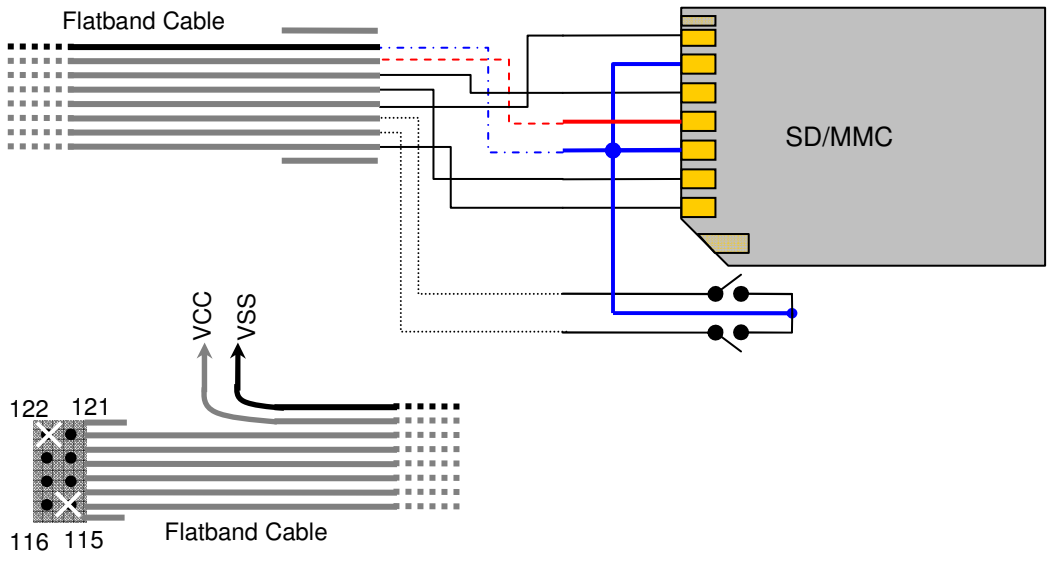
Connector X3 (Pin 71 – 100):

Pin	Description	Function
82	P00_7*	Chipselect (dot of left 7 segment display shows activity = CS)*
83	P01_0	Card insert switch
84	P01_1	Write protection switch
85	P01_2/SIN3*	SPI Input (MCU)*
86	P01_3/SOT3*	SPI Output (MCU)*
87	P01_4/SCK3*	SPI Clock (MCU)*
88	VCC*	VCC connected to MMC/SD card*
89	VSS	VSS connected to MMC/SD card

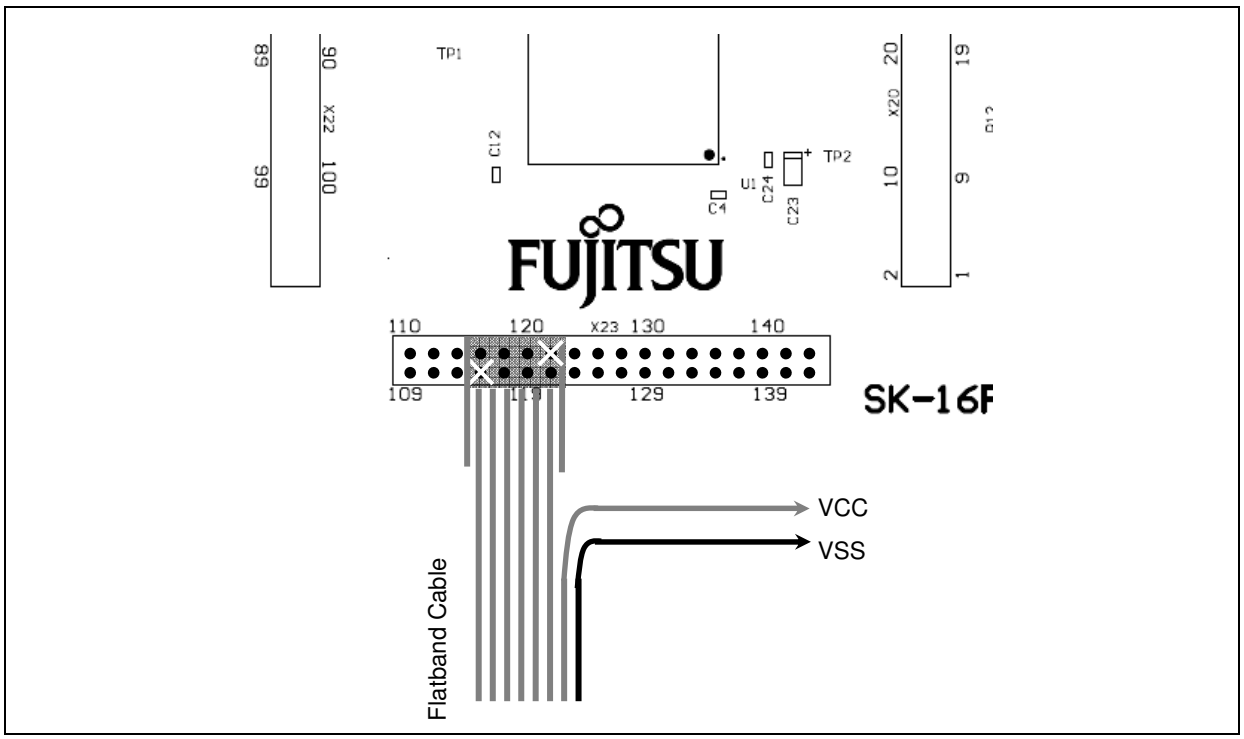
* 3,3V

2.3 Sample Connection for SK-16FX-144PMC-USB (MB96330 Series)

To connect a MMC/SD Card to the Starterkit, UART 3 can be used to be pin port compatible to the SK-16FX-Euroscope Sample**. Do not forget to switch the MCUVCC voltage to 3,3V via jumper JP10**. Connect a MMC/SD Card slot with a flatband cable as it is shown below. On the other side a normal pin header connector can be used. For a stable connection the flatband cable has to be not longer then 20cm.



At the Starterkit use connection header X3. Connect the adapter cable from pin 115 to 122, pin 115 and 122 are not used in this example:



For connections details have a look in the following table. It shows pin-number at the MCU, function of the pin, which port is used and which pin is connected to the MMC/SD Card. Port 0 is also used for the right seven segment display. Pin *P00_P7* is the dot in the display and shows cable select activity of the MMC/SD card.

Connector X3 (Pin 109 – 144):

Pin	Description	Function
116	P00_7*	Chipselect (dot of left 7 segment display shows activity = CS)*
117	P01_0	Card insert switch
118	P01_1	Write protection switch
119	P01_2/SIN3*	SPI Input (MCU)*
120	P01_3/SOT3*	SPI Output (MCU)*
121	P01_4/SCK3*	SPI Clock (MCU)*
VCC*	VCC*	VCC connected to mmc/sd card*
VSS	VSS	VSS connected to mmc/sd card

* 3,3V

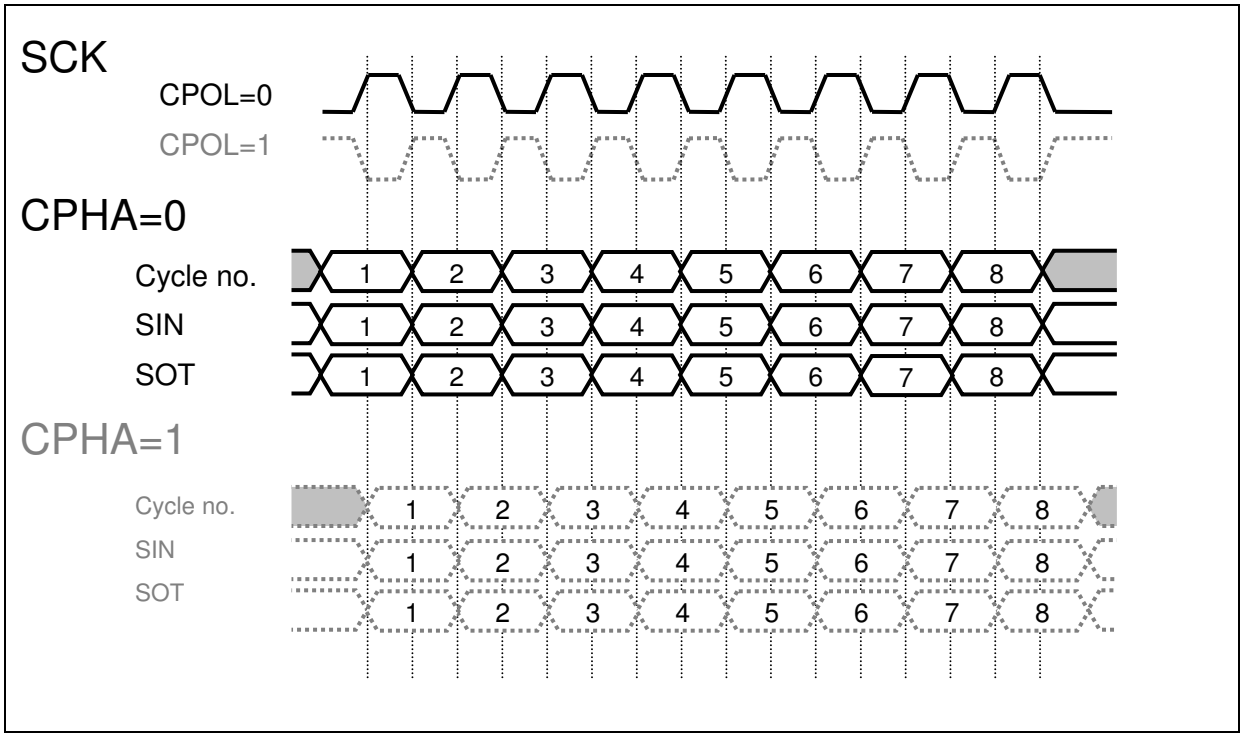
2.4 MMC/SD Card Hardware Configuration

All following settings are customized for the MCU on which the application is running: In this example for the MB96300 series. The section describes the SPI interface mode, UART settings, Port settings and timer interrupt settings which are required to interface the MMC/SD Card.

2.4.1 SPI Mode

The SPI interface is a synchronous serial communication which has a data input, a data output and a clock signal generated by the MCU. For MMC/SD Cards SPI mode 0 is used. For SPI mode 0, CPOL and CPHA are cleared (CPOL = 0, CPHA = 0). CPOL configures the clock signal: normal or inverted. CPHA configures the clock delay.

SPI Communication modes



2.4.2 UART Configuration

First the input and output direction registers has to been set for UART 3. The input pin of the UART has to be configured with an internal pull-up:

```
DDR01_D2=0;          // Direction Input
PIER01_IE2=1;        // Input Enable
PUCR01_PU2=1;        // Set Pullup
```

Next the baud rate is configured. The communication for SPI communications with a MMC/SD Card are defined with 400KBaud. At a clock speed of 48MHz the baud rate register has to be 48MHz/400KBaud = 120.

The SPI mode can be set in the registers ESCR3 and ECCR3. ESCR3 has to be 0x00, because for normal clock mode CPOL = SCES bit has to be cleared. In ECCR3, CPHA = SCDE bit enables or disables the clock delay, the master slave bit has to be configured as master (MS=0). No clock delay is used, so ECCR3 has to be 0x00.

ESCR3 Register:

SCES	CCO	SIOP	SOPE	LBL0	LBL1	LBD	LBIE	Value
0	0	0	0	0	0	0	0	0x00

ECCR3 Register:

INV	LBR	MS	SCDE	SSM	BIE	RBI	TBI	Value
0	0	0	0	0	0	0	0	0x00

To enable data transfers, the reception and transmission has to be enabled. This can be done in SCR3 register:

SCR3 Register:

PEN	P	SBL	CL	AD	CRE	RXE	TXE	Value
0	0	0	0	0	0	1	1	0x03

The UART has to be configured in synchronous mode with clock signal. MD0 = 0 and MD1 = 1 configures the UART as synchronous. SCKE = 1 enables the clock signal and SOE = 1 enables the serial output driver.

SMR3 Register:

MD1	MD0	OTO	EXT	RESET	UPCL	SCKE	SOE	Value
1	0	0	0	0	0	1	1	0x83

In SSR3 register MSB/LSB order is defined. MSB has to be first, so BDS is set.

SSR3 Register:

PE	ORE	FRE	RDRF	TDRE	BDS	RIE	TIE	Value
0	0	0	0	0	1	0	0	0x04

The configuration of the UART is found in *hwsupport.c* as void `InitSPI3()`.

UART Configuration in *hwsupport.c*

```
// file: hwsupport.c
...
void InitSPI3( void )
{
  // set portpin SIN3 to input
  DDR01_D2=0;          // Direction Input
  PIER01_IE2=1;        // Input Enable
  PUCR01_PU2=1;        // Set Pullup
  BGR3 = 120;          // Baud Rate 400KBaud
  ESCR3 = 0x00;        // SCES = 0 => CPOL = 0
  ECCR3 = 0x00;        // SCDE = 0 => CPHA = 0
  SCR3 = 0x03;         // reception and transmission enable
  SMR3 = 0x83;         // Mode 2, SCLK enable, SOT enable
  SSR3 = 0x04;         // MSB first, no interrupts
}
...
```

In *mmc.h* following UART 3 registers are defined:

```
#define SSR_RDRF       SSR3_RDRF
#define RDR            RDR3
#define SSR_TDRE       SSR3_TDRE
#define TDR            TDR3
```

2.4.3 Port Definitions

First the I/O Ports have to be initialized. SOT3 and SIN3 pin are configured with the UART (chapter 2.4.2). Before the switch ports can be used they have to be configured as input and input has to be enabled. Also the switch ports are connected with internal pull-up's. The cable select signal has to be declared as output.

Initalisations of the I/O ports in *hwsupport.c*:

```
void InitPortsSDMMC( void )
{
  DDR01_D1 = 0;                // write protect switch - port direction input
  PIER01_IE1 = 1;              // write protect switch - input enabled

  DDR01_D0 = 0;                // sdcard switch - port direction input
  PIER01_IE0 = 1;              // sdcard switch - input enabled

  DDR00_D7 = 1;                // CS - Signal for sdcard - port direction output
}
```

Next the ports have to be defined in the *mcc.h* file. `ECCR3_TBI` is the Transmission Bus Idle bit, which is cleared if a transmission is ongoing. The definition `SELECT()` and `DESELECT()` defines setting or clearing the cable select signal. The pin can only be toggled if no transmission is ongoing. The line `while(ECCR3_TBI == 0); PDR00_P7 = 0` waits for transmitting complete and clears the CS pin. The same is done for setting the pin.

```
#define SELECT()       while(ECCR3_TBI == 0); PDR00_P7 = 0   /* MMC CS = L */
#define DESELECT()     while(ECCR3_TBI == 0); PDR00_P7 = 1   /* MMC CS = H */
```

The switches for write protection and card insert are located at the same ports. In the example at Port `PDR01` pin 1 and 2 of the port.

```
#define MM_SOCKPORT    PDR01                /* Socket contact port */
#define MM_SOCKWP      0x02                 /* Write protect switch */
#define MM_SOCKINS     0x01                 /* Card detect switch */
```


All Definitions of the file *mcc.h*

```
// file: mcc.c
...
/* Control signals for Fujitsu MB96F300 series */
#define SELECT()       while(ECCR3_TBI == 0); PDR00_P7 = 0   /* MMC CS = L */
#define DESELECT()     while(ECCR3_TBI == 0); PDR00_P7 = 1   /* MMC CS = H */

#define MM_SOCKPORT    PDR01                  /* Socket contact port */
#define MM_SOCKWP      0x02                   /* Write protect switch */
#define MM_SOCKINS     0x01                   /* Card detect switch */

/* Wait functions for  Fujitsu MB96F300 series */
#define waitshort()    __wait_nop()
#define waitlong()     __wait_nop();__wait_nop();__wait_nop();__wait_nop()

/* SPI for Fujitsu MB96F300 series */
#define SSR_RDRF       SSR3_RDRF
#define RDR            RDR3
#define SSR_TDRE       SSR3_TDRE
#define TDR            TDR3
```

2.4.4 Timer Definitions

The MMC/SD Card I/O driver needs a Reload Timer Interrupt for timeout functionality and checking if a card was inserted.

The timer initialization is handled in *hwsupport.c*. The prescaler is set to CLKP1 / 2^6 , which means 48MHz/64 = 750KHz. The timer operates in reload mode (TMCSR0_UF = 1). To reach a reload timer interrupts, which is called every 10ms (100Hz), reloadvalue = (CLKP1 / prescaler)/(wanted interval) = 7499.

Timer Initializations in *hwsupport.c*

```
void InitTimer0( void )
{
        TMCSR0=0x0800; // prescaler
        TMCSR0_UF=1;   //timer operates in reload mode
        TMCSR0_RELD=1;         // Reload value 750KHz/100Hz - 1=7499
        TMRLR0 = 7499;
        //timer enabled
        TMCSR0_CNTE=1;
        //start timer
        TMCSR0_TRG=1;
        //enable the timer interrupt
        TMCSR0_INTE=1;
}
```

The initialisation of the interrupt has to be done in *vectors.c*. The interrupt number is specific for the MB96F348H microcontroller.

Timer Interrupt Initialisations in *vectors.c*

```
...
ICR = (51 << 8) | 4;   // RLT0
...
__interrupt void ReloadTimer0_IRQHandler( void );
...
#pragma intvect ReloadTimer0_IRQHandler 51
...
```

The interrupt routine is handled in hwsupport.c. Necessary for MMC/SD card support is the call of `disk_timerproc()`. This is be done every 10ms.

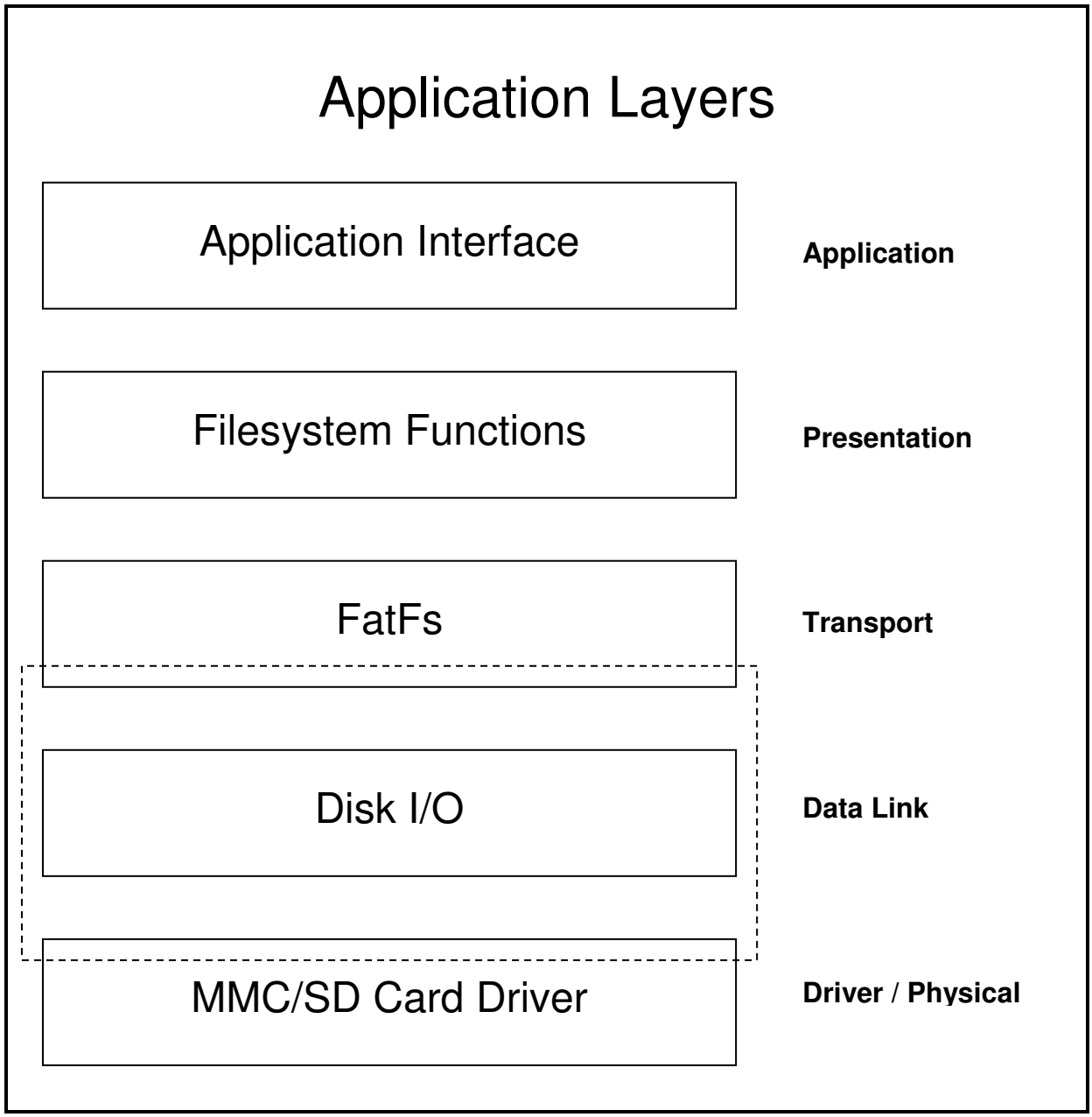
ISP of ReloadTimer0 in *hwsupport.c*

```
__interrupt void ReloadTimer0_IRQHandler( void )
{
 if( TMCSR0_UF) {
     // Reload Timer interrupt request
     TMCSR0_UF=0; // clear interrupt flag
     disk_timerproc(); // in mcc.c, every 10ms
     timer++;
     if (timer == 100) // every second
     {
       uptime++;
       timer=0;
     }
     if (timeout) timeout--; // timeoutcounter
 }
}
```

3 Software

DESCRIPTION OF DRIVER AND FILESYSTEM IMPLEMENTATION

The software part describes how the MMC/SD Card is implemented from physical layer to presentation layer. The Application should only use the presentation layer, but it is also possible to use the transport layer. To add useful functions, a new presentation layer “Filesystem Functions” was added between the FatFs module and the main application.

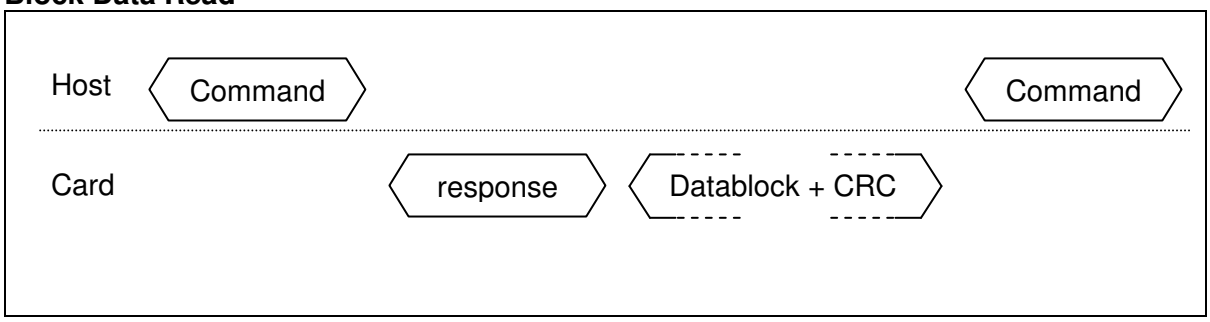


3.1 MMC/SD Card Driver

3.1.1 Protocol

The MMC/SD Card Protocol in this project is based on SPI communication. The Protocol has a command data structure. A Command is send and gets a response or data. To speed up multiple data read cycles the MMC/SD Card supports also a 512Byte block read. In the following example a block read procedure is shown.

Block Data Read

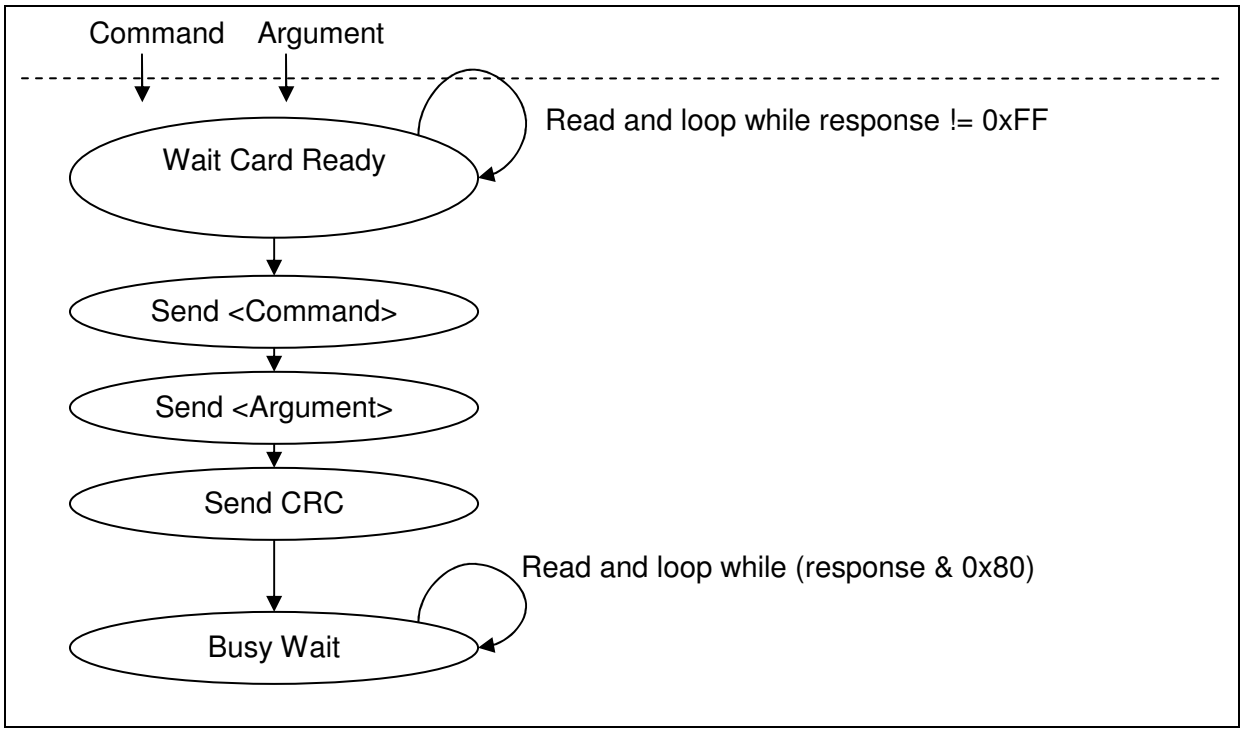


3.1.2 Commands

A command is 6 bytes long and gets a positive response if it was accepted. The command sequence has a command with a size of 1 byte followed with 4 bytes argument and 1 CRC byte.

Command	Argument				CRC
CMD[7..0]	ARG[31..24]	ARG[23..16]	ARG[15..8]	ARG[7..0]	CRC

If a command is initiated, it waits first for card ready. After the card is in idle mode, the command sequence is send. After a positive response, the card is ready for the next command or while in read mode for sending the requested data or data block.



A list of in this project implemented commands is listed below:

Commands Table

Command	Value (SPI)	Description
CMD0	(0x40+0)	GO_IDLE_STATE
CMD1	(0x40+1)	SEND_OP_COND (MMC)
ACMD41	(0xC0+41)	SEND_OP_COND (SDC)
CMD8	(0x40+8)	SEND_IF_COND
CMD9	(0x40+9)	SEND_CSD
CMD10	(0x40+10)	SEND_CID
CMD12	(0x40+12)	STOP_TRANSMISSION
ACMD13	(0xC0+13)	SD_STATUS (SDC)
CMD16	(0x40+16)	SET_BLOCKLEN
CMD17	(0x40+17)	READ_SINGLE_BLOCK
CMD18	(0x40+18)	READ_MULTIPLE_BLOCK
CMD23	(0x40+23)	SET_BLOCK_COUNT (MMC)
ACMD23	(0xC0+23)	SET_WR_BLK_ERASE_COUNT (SDC)
CMD24	(0x40+24)	WRITE_BLOCK
CMD25	(0x40+25)	WRITE_MULTIPLE_BLOCK
CMD55	(0x40+55)	APP_CMD
CMD58	(0x40+58)	READ_OCR

3.1.3 Disk I/O Interface

The MMC/SD Card Driver adapts to the disk I/O functions. The disk I/O functions having the function to link the data through standard functions - which are defined in the data link layer – to the driver. Following functions have to be implemented in the driver sections:

`DSTATUS disk_initialize ( void )`
This function is assigned to handle the disk initialization progress and returns the disk status after the initialization progress.

`DSTATUS disk_status ( void )`
This part returns the current disk status.

`DRESULT disk_read ( BYTE *buff, DWORD sector, BYTE count )`
The functions reads data from a storage. `*buff` is the pointer to the memory in which the data is stored, `sector` is the sector position from which data is read and `count` is the length in sectors, which shall be read.

`DRESULT disk_write ( BYTE *buff, DWORD sector, BYTE count )`
The functions writes data to a storage. `*buff` is the pointer to the memory in which the data is stored, `sector` is the sector position from which data is written and `count` is the length in sectors, which shall be written.

`DRESULT disk_ioctl ( BYTE ctrl, void *buff )`
This function handles Miscellaneous Functions, which are most specific on the storage which is used.

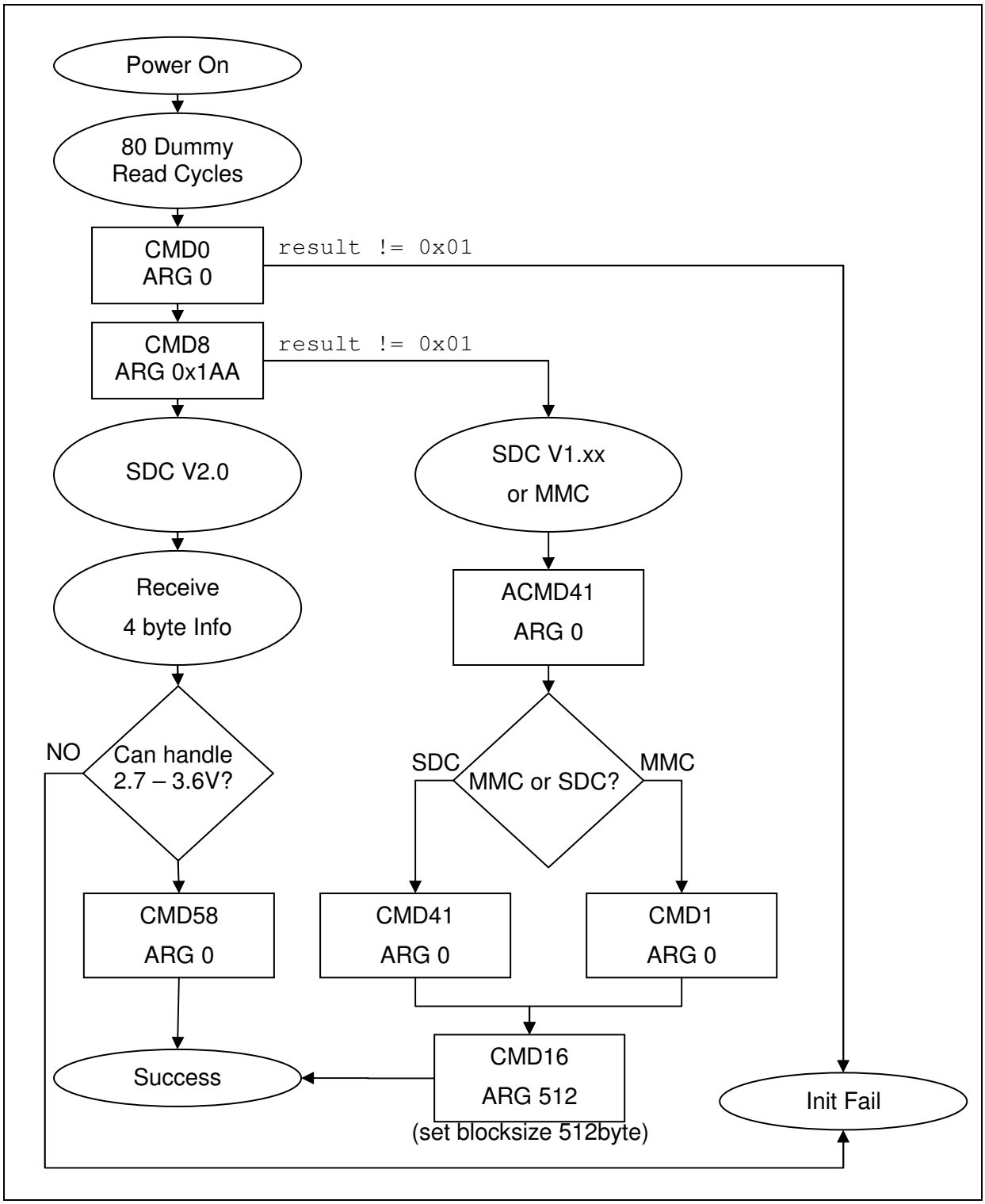
The different types of I/O-Controls can be found in *diskio.h*. Here are some definitions for standard I/O-operations and also some for MMC/SD Card specific.

Definitions for disk_ioctrl() in *diskio.h*

```
/* Command code for disk_ioctrl() */
/* Generic command */
#define CTRL_SYNC               0      /* Mandatory for write functions */
#define GET_SECTOR_COUNT        1      /* Mandatory for only f_mkfs() */
#define GET_SECTOR_SIZE         2
#define GET_BLOCK_SIZE          3      /* Mandatory for only f_mkfs() */
#define CTRL_POWER              4
#define CTRL_LOCK               5
#define CTRL_EJECT              6
/* MMC/SDC command */
#define MMC_GET_TYPE            10
#define MMC_GET_CSD             11
#define MMC_GET_CID             12
#define MMC_GET_OCR             13
#define MMC_GET_SDSTAT          14
```

3.1.3.1 Disk Initialisation

The following diagram shows how the initialisation process works. After Power On and 80 Dummy Cycles, the microcontroller tries to get the card in idle mode and gets the type of the card which was inserted. After all the card will ne initiated for following operations.



3.1.3.2 Disk Status

The Disk Status is realized with a status variable. The status is permanent updated with a reload timer, that executes a status update every 10ms. The disk status has 3 status bits:

Disk status bits, definitions in file *diskio.h*

```
#define STA_NOINIT          0x01  /* Drive not initialized */
#define STA_NODISK          0x02  /* No medium in the drive */
#define STA_PROTECT         0x04  /* Write protected */
```

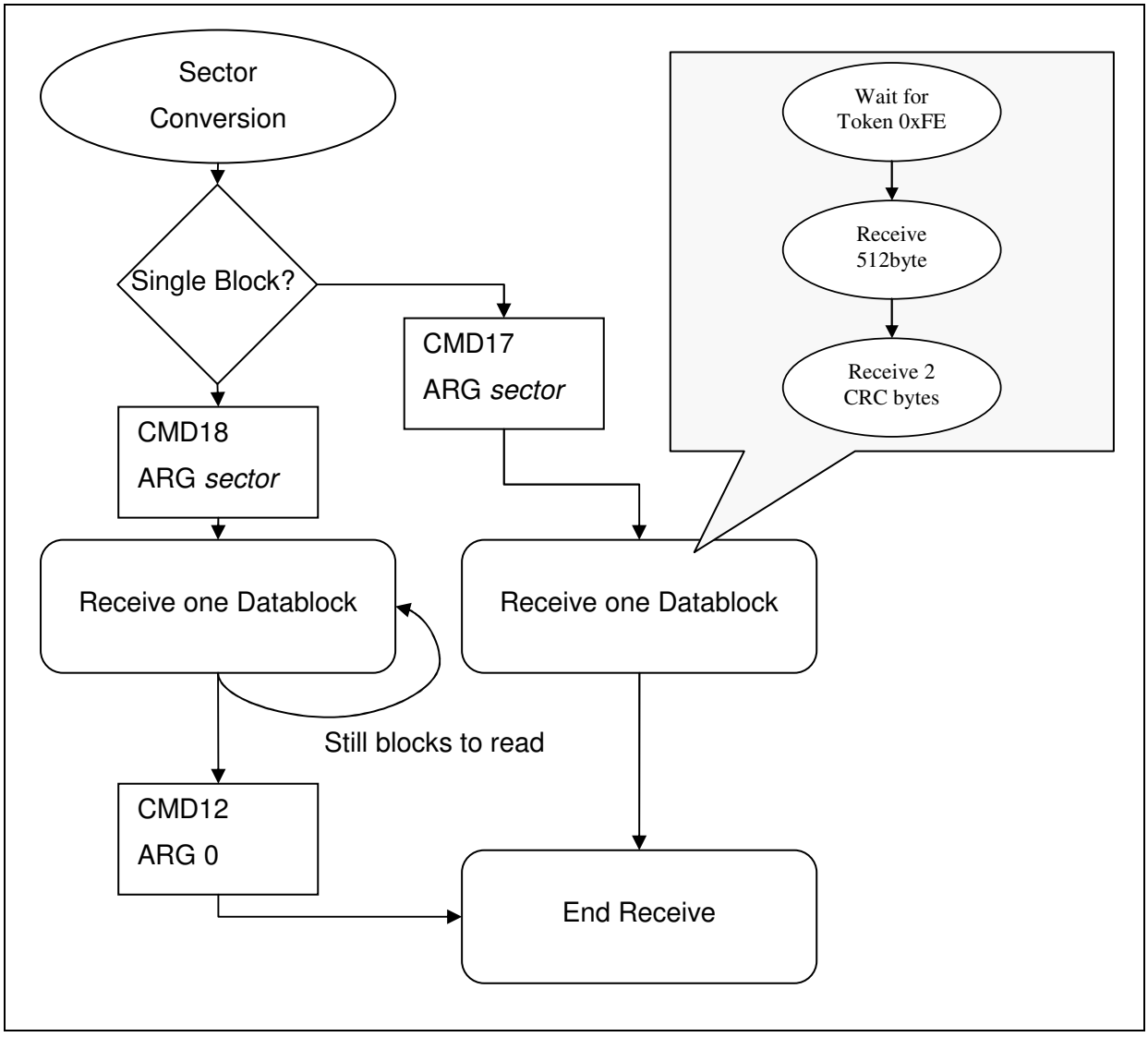
STA_NOINIT is handled during the initialization progress

STA_NODISK is handled with the timer. The status is read directly from the disk insert switch

STA_NOINIT is handled with the timer. The status is read directly from the write protect switch

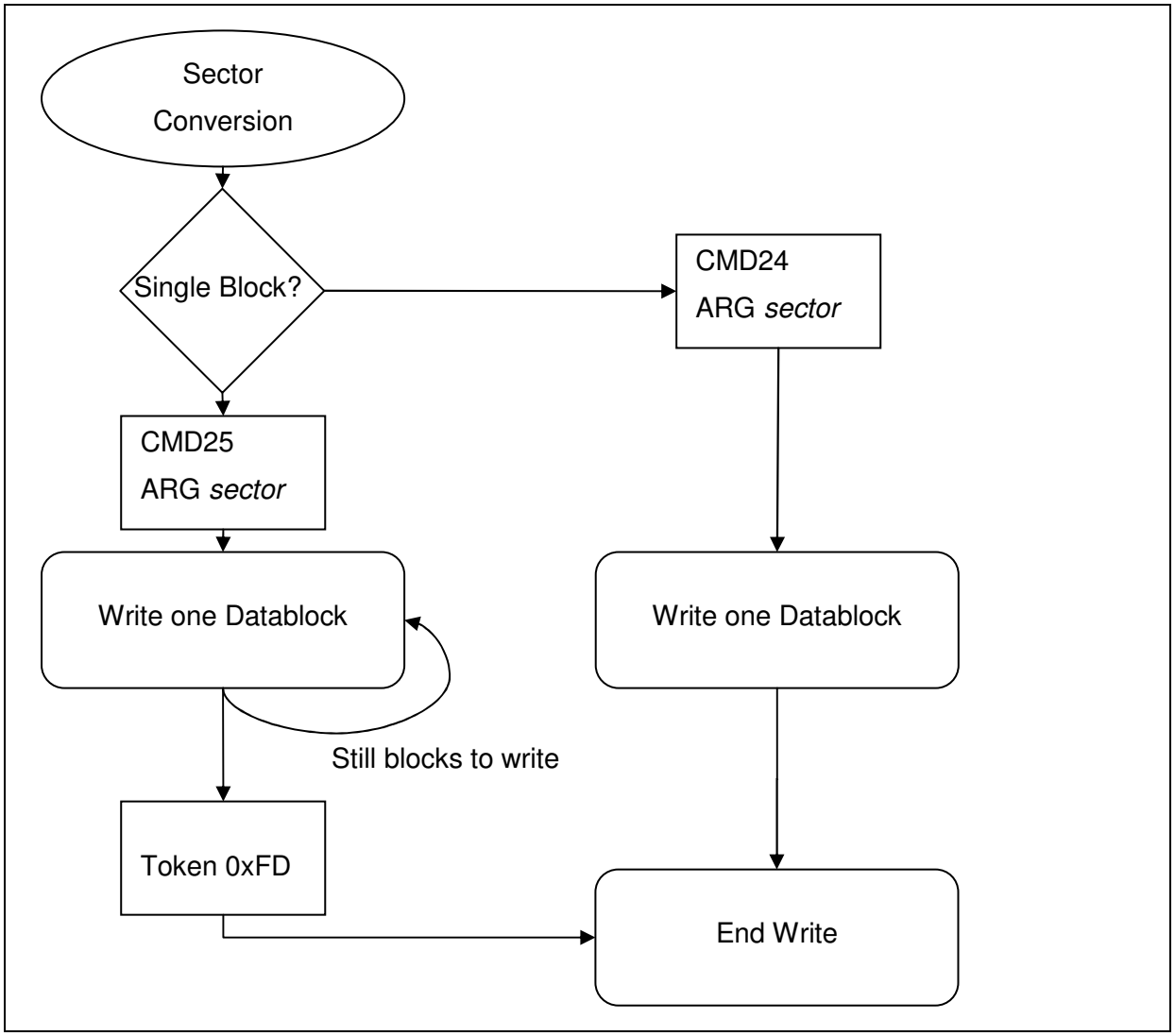
3.1.3.3 Disk Read

The Disk Read function has to do a conversion from sectors into LBA sectors. After the conversion it has to been checked if only one block or more then one blocks are read.



3.1.3.4 Disk Write

The Disk Write function has to do a conversion from sectors into LBA sectors. After the conversion it has to been checked if only one block or more then one blocks are written.



3.1.3.5 Disk IO Ctrl

For more information see chapter Disk I/O

3.1.3.6 Optimization/Minimization

In read/write datablock functions there is the need of a CRC-CIITT 16Bit checksum. For more speed but also secure transfers, the CRC algorithm is realized with a 512 byte table. If more flash is needed there is the possibility to disable CRC check in the *mcc.c* file:

```
#define USE_CRC          1   //by default 1 can be switched to 0
```

3.2 FatFs (FAT Filesystem) & Disk IO Module (supports FAT, FAT16, FAT32)

IMPLEMENTATION OF FAT FILESYSTEM

The FAT Filesystem is based on FAT Filesystem Module of Elm Chang (http://elm-chan.org/fsw/ff/00index_e.html). The Version which is used in this project is "R0.06" from April 2008. For detailed information have a look on the website. Available features are listed below:

3.2.1 FatFs/Tiny-FatFs module provided functions

`f_mount`	Register or Unregister a Work Area
`f_open`	Open or Create a File
`f_close`	Close a File
`f_read`	Read File
`f_write`	Write File
`f_lseek`	Move File R/W Pointer
`f_truncate`	Truncate File
`f_sync`	Flush Cached Data
`f_opendir`	Open a Directory
`f_readdir`	Read a Directory Item
`f_getfree`	Get Free Clusters
`f_stat`	Get File Status
`f_mkdir`	Create a Directory
`f_unlink`	Remove a File or Directory
`f_chmod`	Change Attribute
`f_utime`	Change Timestamp
`f_rename`	Rename/Move a File or Directory
`f_mkfs`	Create a File System on the Drive
`f_forward`	Forward file data to the stream directly
`fgets`	Read a string
`fputc`	Write a character
`fputs`	Write a string
`fprintf`	Write a formatted string

3.2.2 Disk I/O Interface

`disk_initialize`	Initialize disk drive
`disk_status`	Get disk status
`disk_read`	Read sector(s)
`disk_write`	Write sector(s)
`disk_ioctl`	Control device dependent features
`get_fattime`	Get current time

3.2.3 Optimization/Minimization

For minimizations in the fat filesystem, some functions can be disabled. In *ff.h* some optimization options can be found:

```
#define _FS_READONLY  0
```

By default this is set to 0. Setting `_FS_READONLY` to 1 defines read only configuration. This removes writing functions, `f_write`, `f_sync`, `f_unlink`, `f_mkdir`, `f_chmod`, `f_rename`, `f_truncate` and useless `f_getfree`.

```
#define _FS_MINIMIZE  0
```

By default this is set to 0. The `_FS_MINIMIZE` option defines minimization level to remove some functions:

- 0: Full function.
- 1: `f_stat`, `f_getfree`, `f_unlink`, `f_mkdir`, `f_chmod`, `f_truncate` and `f_rename` are removed.
- 2: `f_opendir` and `f_readdir` are removed in addition to level 1.
- 3: `f_lseek` is removed in addition to level 2.

To use less resources, *tff.c* and *tff.h* (tiny fat filessytem) can be used instead of *ff.c* and *ff.h*. For more information have a look at the developers site:

http://elm-chan.org/fsw/ff/00index_e.html

3.3 Filesystem Functions

This Module adds some features and shows some examples for the FAT Filesystem. The module acts as a presentation layer or API to use FAT functionality from the main application. It is also possible to use directly the FAT Filesystem module without this module.

For a better directory handling, reading or writing files, some filesystem functions were added with this module. It is possible to change the current path, doing relative path or absolute path operations. Most of the functions are written for absolute directories. To handle absolute and relative directories, all absolute directory only functions are static and can only used in this module and had been added by a public function, which does a conversion before running the absolute function. For example `BOOL absReadFile(char *filename)` which handles absolute directories only, has also a public function `BOOL readFile(char *filename)` which handles absolute and relative directories.

3.3.1 Features Overview

- Error handling and error display over UART
- Directory- and File-String Functions
- Examples for reading, writing and append files
- Functions for copy, delete, rename and move files and folders
- Optimized String Functions to speed up processes and use less memory

3.3.2 Function Overview (*fsfunctions.c*):

3.3.2.1 Error Handling

- **`FRESULT CatchErr(char *description, FRESULT exception );`**
  Throw out a error message over UART and returns the error code as `FRESULT`

3.3.2.2 File/Path String Operations

- **`void getFullFileName( char *strOut, char *relFile );`**
  Converts a relative file or directory path to a absolute filename or directory path. The result is stored in `strOut`.

  Example:
  Current path is */test*. In this directory a file *file.txt* is located. `getFullFileName` returns a string */test/file.txt*

- **`void filenameFromFullfilename(char *outFile, char *inFullfilename);`**
  Gets the filename from a fullfilename and stores it in the string **`outFile`**.

  Example:
  The fullfilename is */test/file.txt*. `filenameFromFullfilename` returns a string *file.txt*

- **`void pathFromFullfilename(char *outDir, char *inFullfilename);`**
  Gets the path from a fullfilename and stores it in the string **`outDir`**.

  Example:
  The fullfilename is */test/file.txt*. `pathFromFullfilename` returns a string */test*

- **`BYTE splitParameters(char *inputStr);`**
  splits the parameters stored in `inputString` separated by space into `params[]`. Returns the number of parameters. Maximum of available parameters and parametersize is stored in the headerfile.

3.3.2.3 Workspace Operations

- **`void mount_workspace(BYTE drive);`**
  mounts a workspace to use filesystem functions. (drive: 0 = MMC, 1 = USB)
- **`void unmount_workspace(BYTE drive);`**
  unmounts a workspace. (drive: 0 = MMC, 1 = USB)

3.3.2.4 File Operations

- **`FRESULT fileopen(FIL *fp, const char *mytextfile, BYTE mode);`**
  does the same like `f_open` in the FatFs module but uses relative and absolute filenames. For file close use `f_close(FIL *fp)`
- **`BOOL readFile( char *mytextfile );`**
  this is an example how to read the file `mytextfile` to the UART
- **`BOOL appendFile( char *mytextfile, char *mytext );`**
  this is an example how to add a text `mytext` to file `mytextfile`. If `mytext` is empty, `appendFile` asks for text input
- **`BOOL writeFile( char *mytextfile, char *mytext );`**
  this is an example how to write a text `mytext` to file `mytextfile`. If `mytext` is empty, `appendFile` asks for text input
- **`void copyFile(char * from, char * to);`**
  copy file `from` to `to`. Works only on the same drive not from one drive to another
- **`BOOL existsFile( char *myfile );`**
  returns `TRUE` if file `myfile` does exists, otherwise `FALSE`

3.3.2.5 Directory Operations

- **`void makeDir(char *newdir);`**
  creates a new Dir `newdir`
- **`void changeDir( char *newDir );`**
  changes current directory to the specified dir or goes a directory backward by entering ".." as `newDir`
- **`void dirUpper( void );`**
  does the same `like changeDir("..")`

3.3.2.6 Filesystem Operations

- **`BOOL delFile( char *thefile );`**
  deletes the file `thefile`

- **BOOL moveFileFolder(char *file1, char *location);**
  moves file or folder to a new location
- **BOOL renameFileFolder(char *file1, char *file2);**
  renames file or folder
- **void scanFiles (char* path);**
  this is an example how to scan files in specified path and list them on UART
- **void diskSpace(void);**
  this is an example how to show the free and available disk space on UART
- **void formatDisk(BYTE drive);**
  this is an example how to format a disk drive, asks for confirmation over UART. (drive: 0 = SD/MMC, 1 = USB)

3.3.2.7 Global Variables (*fsfunctions.c*):

```
extern FIL ffile;
/* global file for file i/o routines */


extern FATFS fatfs[2];
/* global filesystems for drive 0/1 */


extern currentdir[MAX_DIRLEN];
/* stores the current directory */


extern char params[MAX_PARAMS][MAX_PARMSIZE];
/* stores the parameters for a command */


extern BOOL batch;
/* if the command line reads from a batch file, batch is set */
```

3.3.3 Function Overview (*strfunc.c*):

```
unsigned char cmpFromStart(const char *s1, const char *s2);
/* optimized for speed; compares s1 and s2 from start. If s2 is in s1, function
   returns 0 if successful, returns 1 for not successful */


void putdec2(unsigned long x);
/* puts 2 digits of a value stored in x to the UART */


void addStr(char *s1, const char *s2);
/* adds string s2 to string s1 */


void remLineBreak(char *Str);
/* deletes a line break at the and of a string */


void shiftStr(char *Str, unsigned char offset);
/* shifts a string left “offset” times */


unsigned int fromHex(const char *inputStr, unsigned char * len);
/* gets hex value from a string containing 0x… */
```

```
void toHex(char * outStr, unsigned long n, unsigned char digits, char showzeros);
/* converts a number "n" to a string in hex format (0x…) with the maximum length
   of "digits". If showzeros is 1, zeros at the beginning are shown. */


unsigned int fromDec(const char *inputStr, unsigned char * len);
/* gets decimal value from a string containing a decimal value */


int strEmpty(const char * Str);
/* checks for a string is empty (first character = '\0') */


void fillspacesatend(char *str, unsigned char length);
/* fill space characters at the end of a string. Length defines the total length
   of the string. The number of spaces is length - length of str.
   This function can be used for tabulators for example */


int isNum(char c);
/* returns 1 if '0' <= c <= '9', else returns 0 */


int strIsNum(const char * c);
/* returns 1 if every char is a number, else returns 0 */


int isHex(char c);
/* returns 1 if '0' <= c <= 'F', else returns 0 */


int strIsHex(const char * c);
/* returns 1 if every char is hex and the string begins with "0x", else returns 0
```

3.3.4 Optimization/Minimization

If there is no use for a complex path and file management and also no use for useful string functions, this module can be removed. It is also possible to use the FatFs module directly. All examples in this module like file read or write can be also programmed in an own application. If the command line is used as frontend, this module can't be deleted but it can be minimized in some sections. In *ff.h* some minimizations can be done (see also chapter 3.2.3). This minimizations removing some functions in the FatFs module and also remove some functions in fsfunctions.c which had used the removed functions of *ff.c*.

```
#define MAX_DIRLEN    200        //in file fsfunctions.h
```

By default `MAX_DIRLEN` is 200. If there is no directory structure or the depth of directories is not bigger than 2, `MAX_DIRLEN` can be 30. `MAX_DIRLEN` can be manually chosen by some parameters: `MAX_DIRLEN` = Depth of directories * 15

```
#define MAX_PARMSIZE  50         //in file fsfunctions.h
#define MAX_PARMS     3          //in file fsfunctions.h
```

The maximum length of a parameter and maximum parameters are defined in `MAX_PARMSIZE` and `MAX_PARAMS`. Defaults are 50 for parameter length and a maximum of 3 parameters. If there is no directory structure or the depth of directories is not bigger than 2, `MAX_PARAMS` can be 30.

The RAM usage of the filesystem functions is about

```
MAX_DIRLEN * 4 + 2 * (MAX_FILELEN + 1) + MAX_PARAMS * MAX_PARMSIZE + 20
```

`FILE_LEN` is a const value which is 15.

3.4 Application Interface

Why was a command line chosen? A command line was chosen to doing multiple operations with the filesystem for testing and demonstrating reasons. A command line gives a user more options for testing and using a filesystem while exploring directories or executing filesystem commands. To add own commands makes the command line flexible and gives the programmer a good base for debugging and easily adding features in short time.

3.4.1 Example Command – How to add own commands

It is possible to add own commands by adding in just 3 sections some code. The following note gives an idea how to implement a own function `cmdMyCmd()` which is placed in the file *commands.c* and called from the command line as `mycommand`. The sample command shows how to handle parameters; in this case two parameters which are printed out through the UART. The array `params[]` contains the available parameters, `paramsc` contains the number of parameters which were entered.

Example: `cmdMyCmd` located in *commands.c*

```
int cmdMyCmd( void )

{
        puts("Hello World\n");
        puts("You've entered ");
        puts(paramsc + '0');
        puts(" parameters:\n");
        puts(The first parameter is \"");
        puts(params[0]);
        puts("\"\n");
        if (paramsc > 1)
        {
          puts(The second parameter is \"");
          puts(params[1]);
          puts("\"\n");
        }
        return 0;
}
```

Input/Output in a Terminalprogram:

```
0:/>mycommand myparameter1 myparameter2

Hello World
You've entered 2 parameters:
The first parameter is "myparameter1"
The second parameter is "myparameter2"

0:/>…
```

3.4.2 Command Sections

3.4.2.1 Prototype Section

In this section add the prototype from the sample application. The prototype section is located in the *commands.h* file.

```
/*      COMMAND SECTION */

int cmdHELP(void);

int cmdMKDIR(void);

int cmdDIR(void);

int cmdCD(void);

int cmdRENAME(void);

int cmdWRITE(void);

int cmdAPPEND(void);
```

```
...
int cmdMyCmd(void);
...
```

3.4.2.2 Struct s_command cmds[]

This section is a table with constant data inside. For a command the table has some options which are needed before a command can be executed in the command line. First a line in the table has a command as string. In the example application it is `mycommand`. It is possible to add short information which is displayed in the `help` command. If the command should be a hidden command, the information has to begin with a "!". The next two parameters define how much parameters the command can handle. First the minimum of parameters which are required to execute a command is written, than the maximum parameters which are allowed for the command. At last a pointer to the function is added. The following textbox explains a line which represents an `s_command` record construct:

Struct s_command record

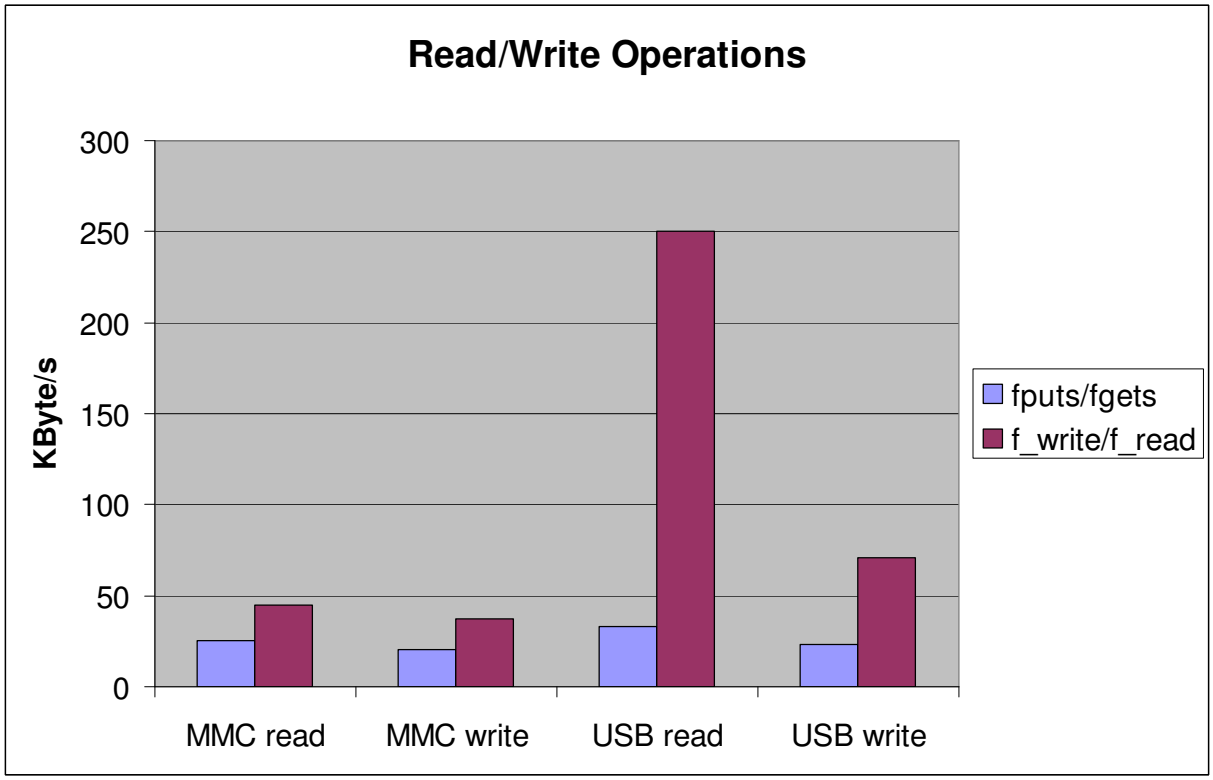
```
{"mycommand", " information", 1,2,cmdMyCmd},
      |               |        | |    |
      |               |        | |    +--- command function
      |               |        | |
      |               |        | +---------- max. parameters
      |               |        +------------ min. parameters
      |               |
      |               +---------------------- short information
      |
      +-------------------------------------- command-string
```

The next code shows the example implemented in the `cmds[]` table in *commands.h*

```
/* Table of commands:
   {String Command, String Description, Min Parameters, Max Parameters, Function to execute}
*/
const struct s_command cmds[] = {
  {"help",      " - displays aviable commands",0,0,cmdHELP},
  {"mkdir",     " </dir1/newdir> - creates new directory \"sub2\"",1,1,cmdMKDIR},
  {"dir",       " </dir1/dir2> - … specified dir",0,1,cmdDIR},
  {"cd",        " <DIR> - changes directory",0,1,cmdCD},
  ...
  {"mycommand", " information", 1,2,cmdMyCmd},
  {"",          "",0,0,cmdIDLE}        // DO NOT REMOVE, HAS TO BE EVER THE LAST ITEM!!
};
```

4 Performance

The Performance was tested with two boards: SK-16FX-Euroscope and SK-16FX-144PMC-USB. In the test 1Mbyte data was written or read. If SK-16FX-144PMC-USB starterkit is used, it is possible to use USB mass storage devices and MMC/SD card support. At SK-16FX-Euroscope starterkit only MMC/SD is supported. The diagram compares MMC/SD cards and USB-Sticks with different write/read functions. `fgets` and `fputs` are string functions which are using `f_write` and `f_read`. If the correct length of bytes to be written or read is known, for performance issues it is better to use `f_write` and `f_read`. In the diagram it can be seen, that the direct `f_write` or `f_read` function is faster then `fputs` or `fgets`. The biggest differences are seen at USB read.



Function	Media/Operation			
	MMC/SD read	MMC/SD write	USB read	USB write
fputs/fgets	25 Kbyte/s	20 Kbyte/s	33 Kbyte/s	23 Kbyte/s
f_write/f_read	43 Kbyte/s	33 Kbyte/s	250 Kbyte/s	71 Kbyte/s

1Kbyte = 1000Byte, 1MByte = 1000Kbyte

1Mbytes in 50 byte blocks are written in this example

5 Appendix

5.1 Related Documents

- Appnote: sk16fx144pmc_usb_host (mcu-an-300243-e-v10)
- Website Elm Chang – FatFs module: http://elm-chan.org/fsw/ff/00index_e.html
- Website Elm Chang – MMC/SD Card: http://elm-chan.org/docs/mmc/mmc_e.html
- Website Ulrich Radig – MMC: http://www.ulrichradig.de/home/index.php/avr/mmc-sd
- SD Card Association - http://www.sdcard.org/home
- Hardware Spec. (SanDisk): http://www.sandisk.com/Oem/Default.aspx?CatID=1416
- Example for MB96F338US: *96330_usb_mass_storage_spi_sd_card-v11.zip*
- Example for MB96F348: *96340_ spi_sd_card-v11.zip*

5.2 Additional Information

Information about FUJITSU Microcontrollers can be found on the following Internet page:

http://mcu.emea.fujitsu.com/