Application Note, V1.0, Oct. 2008

AP16125

XE166 family

SDCard in SPI mode with FAT16

Microcontrollers

Never stop thinking

Edition 2008-10-21

Published by
Infineon Technologies AG
81726 München, Germany

© Infineon Technologies AG 2008.
All Rights Reserved.

LEGAL DISCLAIMER

THE INFORMATION GIVEN IN THIS APPLICATION NOTE IS GIVEN AS A HINT FOR THE IMPLEMENTATION OF THE INFINEON TECHNOLOGIES COMPONENT ONLY AND SHALL NOT BE REGARDED AS ANY DESCRIPTION OR WARRANTY OF A CERTAIN FUNCTIONALITY, CONDITION OR QUALITY OF THE INFINEON TECHNOLOGIES COMPONENT. THE RECIPIENT OF THIS APPLICATION NOTE MUST VERIFY ANY FUNCTION DESCRIBED HEREIN IN THE REAL APPLICATION. INFINEON TECHNOLOGIES HEREBY DISCLAIMS ANY AND ALL WARRANTIES AND LIABILITIES OF ANY KIND (INCLUDING WITHOUT LIMITATION WARRANTIES OF NON-INFRINGEMENT OF INTELLECTUAL PROPERTY RIGHTS OF ANY THIRD PARTY) WITH RESPECT TO ANY AND ALL INFORMATION GIVEN IN THIS APPLICATION NOTE.

Information

For further information on technology, delivery terms and conditions and prices please contact your nearest Infineon Technologies Office (**www.infineon.com**).

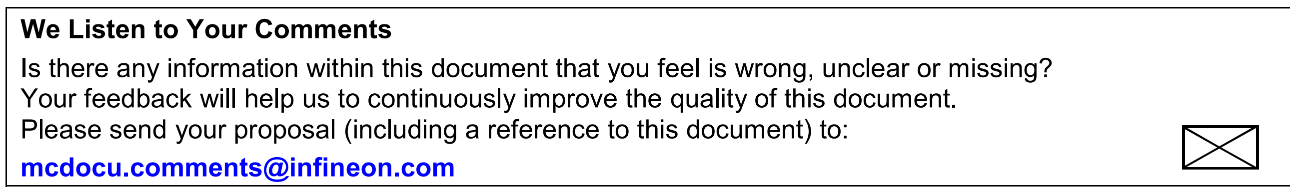
Warnings

Due to technical requirements components may contain dangerous substances. For information on the types in question please contact your nearest Infineon Technologies Office.

Infineon Technologies Components may only be used in life-support devices or systems with the express written approval of Infineon Technologies, if a failure of such components can reasonably be expected to cause the failure of that life-support device or system, or to affect the safety or effectiveness of that device or system. Life support devices or systems are intended to be implanted in the human body, or to support and/or maintain and sustain and/or protect human life. If they fail, it is reasonable to assume that the health of the user or other persons may be endangered.

AP16125		
Revision History:	2008-10	V1.0
Previous Version:	none	

Page	Subjects (major changes since last revision)



We Listen to Your Comments

Is there any information within this document that you feel is wrong, unclear or missing?
Your feedback will help us to continuously improve the quality of this document.
Please send your proposal (including a reference to this document) to:

mcdocu.comments@infineon.com

Table of Contents Page

1 **Introduction - XE166 family** 5

2 **SD Card Overview** 6
2.1 SDCard Specification (Summary) 6
2.1.1 Electrical Interface 6
2.1.2 Power-up 7

3 **The File-system FAT16** 8
3.1 Structure 8
3.1.1 Master Boot Record 8
3.1.2 Boot Record 8
3.1.3 File Allocation Table (FAT) 9
3.1.4 Root Entry 9
3.2 Functionality 9
3.2.1 Get Information about the Disk (Start of FAT, Start of Root Entry) 9
3.2.1.1 Start of FAT 9
3.2.1.2 Start of Root Entry 10
3.2.1.3 Start of Data Area 10
3.2.2 Open and Read a File on Disk 10
3.2.3 Application Hints for the FAT16 Library 11

4 **XE166 Implementation of the SD Card with a FAT16 Library** 12
4.1 Hardware Setup for XE166 12
4.2 Software Setup 13
4.2.1 Power ON (Insertion) 13
4.2.2 Software Reset 13
4.2.3 Initialization 14
4.2.4 How to Support SDCard Ver2 and High Capacity Cards 14
4.2.5 Communication with the SDCard 14
4.2.6 Programmer Hints 15
4.2.6.1 Exact Control of Timing Issues 15
4.2.6.2 Extra ‘waits’ 15
4.2.6.3 Problems with USIC-channel 15
4.2.6.4 Small Stack 15
4.2.6.5 Naming the Files 15

5 **Outstanding Tasks (to be done…)** 16
5.1 The Partitions 16
5.2 Subdirectories 16
5.3 Long Filenames 16
5.4 FAT32 / HCSD-Support 16

1 Introduction - XE166 family

XE166 family - More performance, more Flash, better peripherals

With more than 15 successful years in the microcontroller market place, C166 has set the standard for 16-bit architectures with the highest aggregate volume share of all available 16-bit devices.

With its fast interrupt response and context switching, the C166 family is ideally suited for automotive, industrial, mass storage and wired as well as wireless communications applications.

Compared with the XC166, XE166 delivers more performance, more Flash memory, more RAM, strongly enhanced peripherals and a complete DSP library.

MCU and DSP in a core

Infineon Technologies´ Real Time Signal Controller combines the traditional strengths of a Microcontroller Unit (MCU) to control peripherals with the computing power of Digital Signal Processors (DSP). All in one enhanced XE166 core. Together, the Microcontroller's real-time capability and ease of use and the DSP's mathematical performance and data throughput form a powerful singe-chip solution ideal for many embedded applications.

For detailed technical information about the different derivatives please refer to the XE166 family web pages on the Infineon Internet.

http://www.infineon.com/XE166

2 SD Card Overview

2.1 SDCard Specification (Summary)

This section gives an overview of the basic functionality of an SDCard and its interface. For more detailed information please read the SDCard specification.

This Application Note makes use of the “Simplified physical Layer Specification” document, which can be downloaded at:

http://www.sdcard.org/about/memory_card/pls/Simplified_Physical_Layer_Spec.pdf

2.1.1 Electrical Interface

The SDCard interface provides two different BUS types for interacting with the card:

- SD-Mode
- SPI Mode

The SD mode gives access to all functions of the Card (CRC, High-speed Data transfer, etc.). However implementation is more difficult than the SPI mode.. Furthermore the SPI Bus is already implemented in the USIC of XE166 family.

For this project the SPI Bus is used. Figure 1 shows the SD-Card’s pin assignment.

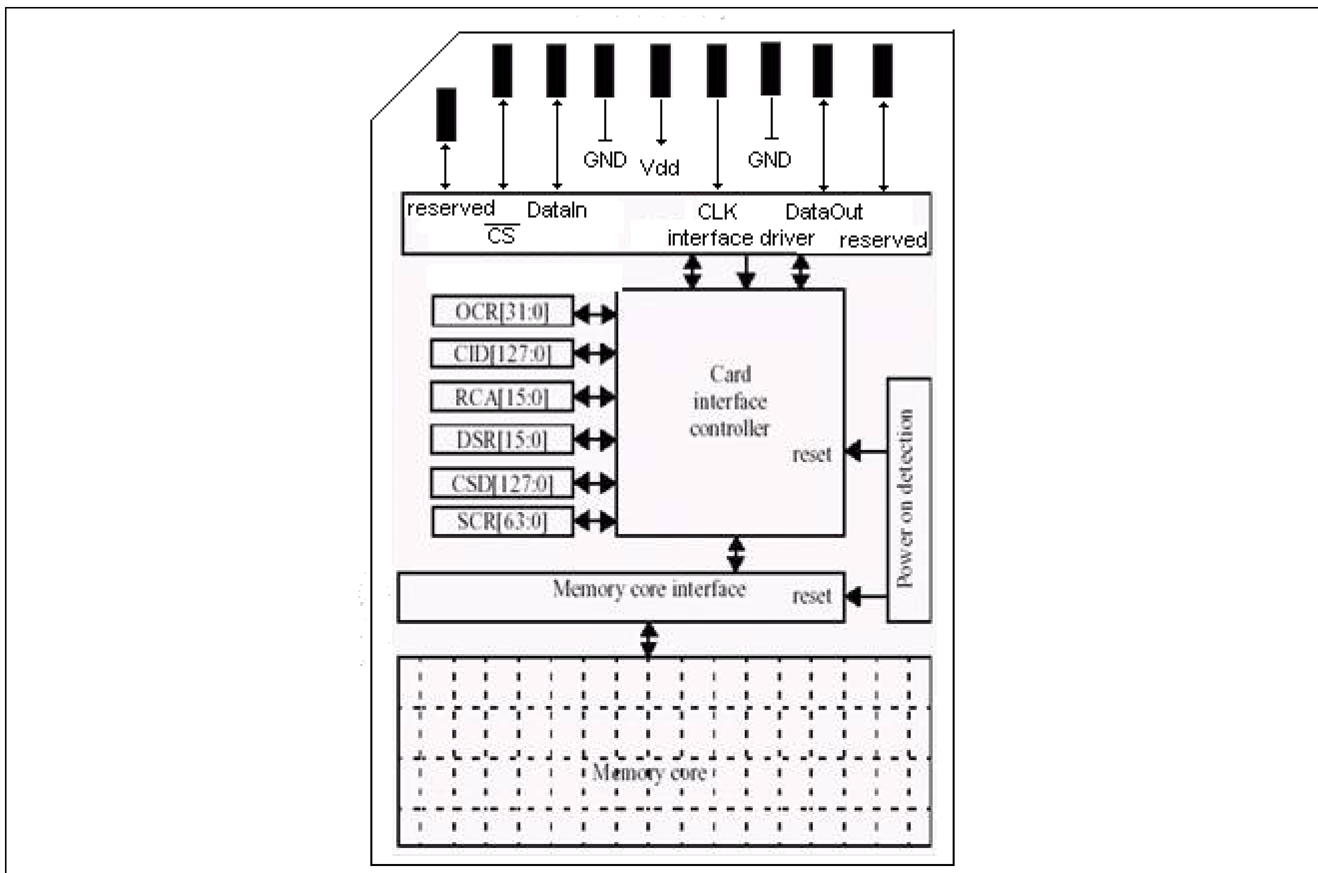


Figure 1 SPI Bus Interface

The SD-Card is to be supplied with a DC-power-source in the Range of 2.7 up to 3.6V. Since the XE166 is supplied with a voltage in this Range (3.3V) there is no need to change this.

In SPI mode only four of the nine pins of the SD Card are of particular interest (except for Vdd and Vss) as detailed below:

Note: In this document the data direction is seen from the cards point of view!

- CS – slave select

  The slave select signal active low is used if more than one SD Card is connected to the bus. With this signal the host can select the device for the communication. To activate the SPI mode the CS signal must be low when sending the CMD0 command. Please refer to the SD-specification for further information.
- CLK – clock

  The clock signal must be generated by the host. When transmitting data from card to host the master must send dummy writes as the card needs the clock cycles for shifting out the data bits. The SD-Card sends and receives Data on the **rising Edge** of the clock. If the clock is inactive the clock-line has to be set high.
- DataIn – host data output – slave data input

  The card receives commands and data from the host via the data input line.
- DataOut – host data input – slave data output

  The card sends respond tokens and data blocks to the host via the data output line.

On all Data-Lines the logical 1 is sent as 3.3V.

The logical 0 is sent as 0V.

Any signal that can float should be pulled low or high properly via a resister. This is a generic design rule on MOS devices. Because DI and DO are normally high, they should be pulled-up. According to SDC/MMC specifications, the recommended value of pull-up registers is from 50k to 100k Ohms.

Note: It is recommended that the SD-Specification is implemented exactly, otherwise the Communication will fail!

2.1.2 Power-up

There is a special power up procedure which needs to be followed for the card to be ready to receive commands in SPI (or SD) mode. Since the card needs a relatively high current during the power-up, the supply should be buffered with a capacitor.

The Card enters the SD mode after connecting to Vdd. After the card has powered up the host needs to send at least 74 clock cycles with high data level. The first command to be sent is CMD0 (Reset).

To enter SPI mode the CS signal has to be low while the host is sending CMD0. If the card detects the SPI mode command it enters the idle-state and sends the Response R1 with LSB set to 1.

For more information about the commands and responses in SPI-Mode, please refer to the SD-specification in chapter 5 of this Application Note, section 4.2.5. This state ignores all commands until ACMD41 (initialize) is received by the card. ACMD41 then starts the card initialization process. After initialization the card is ready to receive other commands and data, and it will be possible to send data.

3 The File-system FAT16

3.1 Structure

A FAT16 formatted device can be divided into 4 sections:

1. Reserved region (containing BPM and boot sector)
2. FAT (File Allocation Table)
3. Root entry
4. Data

3.1.1 Master Boot Record

The Master Boot Record is always the first Block to be read out by a device. It contains information about the way the device is divided into Partitions.

Every Partition contains a Boot Record (see section 3.1.2). The SD-Card has a standard block-size of 512 Bytes per Block, so the Partition entries are to be found at the following Card addresses:

- Partition 1: 0x1BE
- Partition 2: 0x1CE
- Partition 3: 0x1DE
- Partition 4: 0x1EE

All Partition-entries are built in the same Way:

Offset	Size (Byte)	Contains
0x00	1	Bootable (0x80) / non-bootable (0x00)
0x01	3	First Sector of Partition (CHS[1])
0x04	1	Type of Partition (FAT /other)
0x05	3	Last Sector of Partition (CHS)
0x08	4	Start sector of Partition (Offset to MBR in Sectors)
0x0C	4	Number of Sectors

If the Partition-entry is not used, the Partition-type should be set to 0x00.

3.1.2 Boot Record

The boot record contains the Bios Parameter Block (BPM). This contains such information as the number of bytes per sector, sectors per cluster, count of FATs 3.1.3), and so on. These values are required to calculate the start addresses of the root entry.

With the total count of clusters it is also possible to determine which FAT system is implemented on the medium. In addition to the BPM the boot sector contains the jump instruction to boot code and some other information.

[1] Cylinder-Head-Sector Address

3.1.3 File Allocation Table (FAT)

The FAT is a linked list of the cluster of a file. There are mostly two copies of the FAT. Each cluster has an entry in this list. It defines whether the cluster is free, occupied or damaged. If it is occupied it contains the number of the next cluster which belongs to the file, or it marks the End Of File (EOF).

3.1.4 Root Entry

The root entry contains 32 byte file (or directory) entries. It holds the information about the starting cluster of the file, file size, date information, etc. The starting cluster is the starting point of the cluster chain in the FAT. A directory is a file with a special attribute indicating that it is a directory. The contents of this container are 32 byte FAT directory entries.

3.2 Functionality

3.2.1 Get Information about the Disk (Start of FAT, Start of Root Entry)

To perform file operations on a disk it is first necessary to get some information about the file-system, such as where the FAT is, where the start of the data area is, and where the root entry located is. Some useful entries from the BPB block at the boot sector are marked in Figure 2.

- Green: bytes per sector
- Orange: sectors per cluster
- Cyan: reserved sectors
- Blue: number of FATs
- Pink: count of 32byte root entries
- Red: count of sectors of ONE FAT

```
00000000  EB 3C 90 4D 53 44 4F 53 35 2E 30 00 02 20 01 00  ë<.MSDOS5.0.. ..
00000010  02 00 02 00 00 F8 F3 00 3F 00 FF 00 00 00 00 00  .....øó.?.ÿ.....
00000020  00 46 1E 00 00 00 29 61 34 FB 8C 4E 4F 20 4E 41  .F....)a4ûŒNO NA
00000030  4D 45 20 20 20 20 46 41 54 31 36 20 20 20 33 C9  ME    FAT16   3É
```

Figure 2 Boot Sector

Note: Please Note that in the following calculations there is an Offset to the Base of the ***Partition*** *and* ***NOT*** *to the Base of the device!*

3.2.1.1 Start of FAT

The FAT table is directly located after the boot sector. The address can be calculated by multiplying the reserved sectors (cyan) with the bytes per sector (green).

- addressOfFAT1 = reserved sectors * bytes per sector

The second FAT is located after the first FAT. Therefore FATsize (red) has to be added to the reserved sectors).

- addressOfFAT2 = (reserved sectors + FATsize) * bytes per sector

3.2.1.2 Start of Root Entry

The root entry is located after the second FAT. The address of the entry is:

addresOfRoot = addressOfFAT2 + FATsize * bytes per sector

3.2.1.3 Start of Data Area

The data area starts after the last entry of the root directory. The maximum possible number of entries of the root directory is limited by the count of root entries (pink). Therefore most of the FAT16 drives can hold only 512 entries (files or directories) in the root directory.

To get the starting address of the data area, the size of the root directory (in bytes) is added to the start of the root directory:

addressOfData = addressOfRoot + (numnerOfRootEntries * 32)

3.2.2 Open and Read a File on Disk

This example starts at the point where the user supplied the filename of the file to be opened, such as “test8.txt” for example.

The ***first*** step is to find the starting address of the file. The root entry of a file contains the information about the first cluster. With this information the starting address can be calculated. Every entry is a 32byte block. As shown in Figure 3, the first 11 bytes are the filename (8.3 filename format).

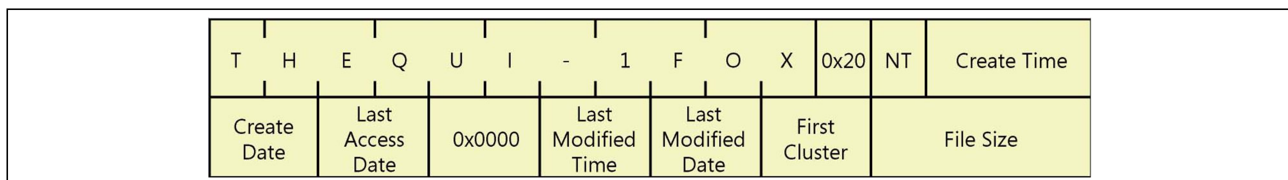


Figure 3 Schematic View of Root Entry

In Figure 4 the first entry is the name of the volume. The next entry holds information about “test8.tx”. The red marked bytes 26 – 27 contain the starting cluster of the file (only in FAT16); e.g. sector 2 (0x 00 02).

The formula for calculating the address of the cluster is very complex, but the basic steps to get the address are as follows:

firstDataSector = reserved sectors + (number of FATs * FAT size) + sectors of root entry

addressOfCluster = firstDataSector * byte per sector

Note: This algorithm is only correct when there is only one partition of the device! For further information please refer to Microsoft’s “fatgen103.doc”. (the FAT specification).

In the ***second*** step the file can be read from disk. The clusters have to be read until the End Of File (EOF) is reached.

The file “test8.txt” is 368640 bytes big. The information about the file size can be found at the green marked bytes 28 – 31 in Figure 4. In this case it is 0x 00 05 A0 00 = 368640bytes = 360kBytes = 23 sectors.

```
0003CE00  53 44 43 41 52 44 31 20 20 20 20 08 00 00 00 00  SDCARD1    .....
0003CE10  00 00 00 00 00 00 10 83 1B 37 00 00 00 00 00 00  .......ƒ.7......
0003CE20  54 45 53 54 38 20 20 20 54 58 54 20 18 98 38 8C  TEST8   TXT .˜8Œ
0003CE30  21 31 21 31 00 00 39 8C 21 31 02 00 00 A0 05 00  !1!1..9Œ!1... ..
0003CE40  54 45 53 54 37 20 20 20 54 58 54 20 18 98 38 8C  TEST7   TXT .˜8Œ
0003CE50  21 31 21 31 00 00 39 8C 21 31 0C 00 00 90 01 00  !1!1..9Œ!1......
```

Figure 4 Root Entry

If the first sector is read out and it is not the EOF, it is necessary to determine which sector is next. The value of the current cluster points to the next cluster. As you can see in Figure 5, the example sector 2 points to sector 3, sector 3 to sector 4, and so on. The last (red marked) sector (0x 00 1C) has the value 0xFFFF which indicates the EOF.

The file size is not required for reading the file, but with that information it is possible to find errors in the file-system. If all sectors have been read and the value of the last cluster does not point to the EOF, the file-system is damaged (That would then require the use of chkdsk to resolve!).

```
00000200  F8 FF FF FF 03 00 04 00 05 00 06 00 07 00 08 00  øÿÿÿ............
00000210  09 00 0A 00 0B 00 10 00 0D 00 0E 00 0F 00 26 00  ..............&.
00000220  11 00 12 00 13 00 14 00 15 00 16 00 17 00 18 00  ................
00000230  19 00 1A 00 1B 00 1C 00 FF FF 1E 00 1F 00 20 00  ........ÿÿ.... .
00000240  21 00 22 00 23 00 24 00 25 00 FF FF 27 00 28 00  !.".#.$.%.ÿÿ'.(.
00000250  FF FF 2A 00 2B 00 FF FF 00 00 00 00 00 00 00 00  ÿÿ*.+.ÿÿ........
00000260  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

Figure 5 FAT

3.2.3 Application Hints for the FAT16 Library

- If a file gets overwritten by a smaller file, the last cluster of the new file contains the data of the old file (i.e. if you open that file with an editor, some bytes from the old file will be seen).
- Only the 8.3 Filename format is supported. There was an implementation of long filenames but is caused errors, and so is disabled in the example code.
- Take into consideration that the memory of the microcontroller is limited. Don't open 10 files at the same time. Close a file if you do not need it any longer.
- The performance of some functions can be improved by changing Baud rates or optimizing the code.
- There is an Implementation for reading out the Partition-Table, but it is *not* tested for many different Cards. It has worked for some cards with a size of 1GB.
- Please note that there are some additional functions implemented in the FAT-library which are NOT tested. The only ***tested*** Functionality is as follows:
    - Read-Out of the root-directory
    - Reading the data of a file in the root-dir
    - Writing Data to a file in the root-dir
    - Deleting a file out of the root-dir
- The example code includes some code to write date of creation, last-modified and last-written entries to the files. For example, there is a procedure called FAT_privateGetTime(void) in the file FAT_LibPrivate.c, that takes the 16-values out of some global variables and creates two 16-bit values (saved in the globals date and time). These values contain the right date and time formatted for standard FAT16-Directory-entries.
  If you wish to use an RTC to create actual files, you just have to save the right time and date into the previously mentioned global variables.
- The Source Ccode is fitted to cards with a block-size of 512 bytes. It will only work with this blocksize. If the blocksize of the card should be different for some reason, you will have to change some code.
- The files to be stored with this code just save ASCII-characters. They are txt-files. To save any other file type some code would have to be written to create the appropriate file header.

4 XE166 Implementation of the SD Card with a FAT16 Library

4.1 Hardware Setup for XE166

The SD Card is connected to the XE166 via the SPI Bus interface. The XE166 is supplied by a 3.3V voltage regulator.

In this example setup the Port 6 pins of the USIC2 channel 0 are used. To ensure the correct start-up a pull up resistor should be connected to the CS line. The Card Detection and Write Protection Signals should also be pulled up and can be connected to any GPIO on the microcontroller. An additional capacitor of 100µF has been used to buffer the supply.

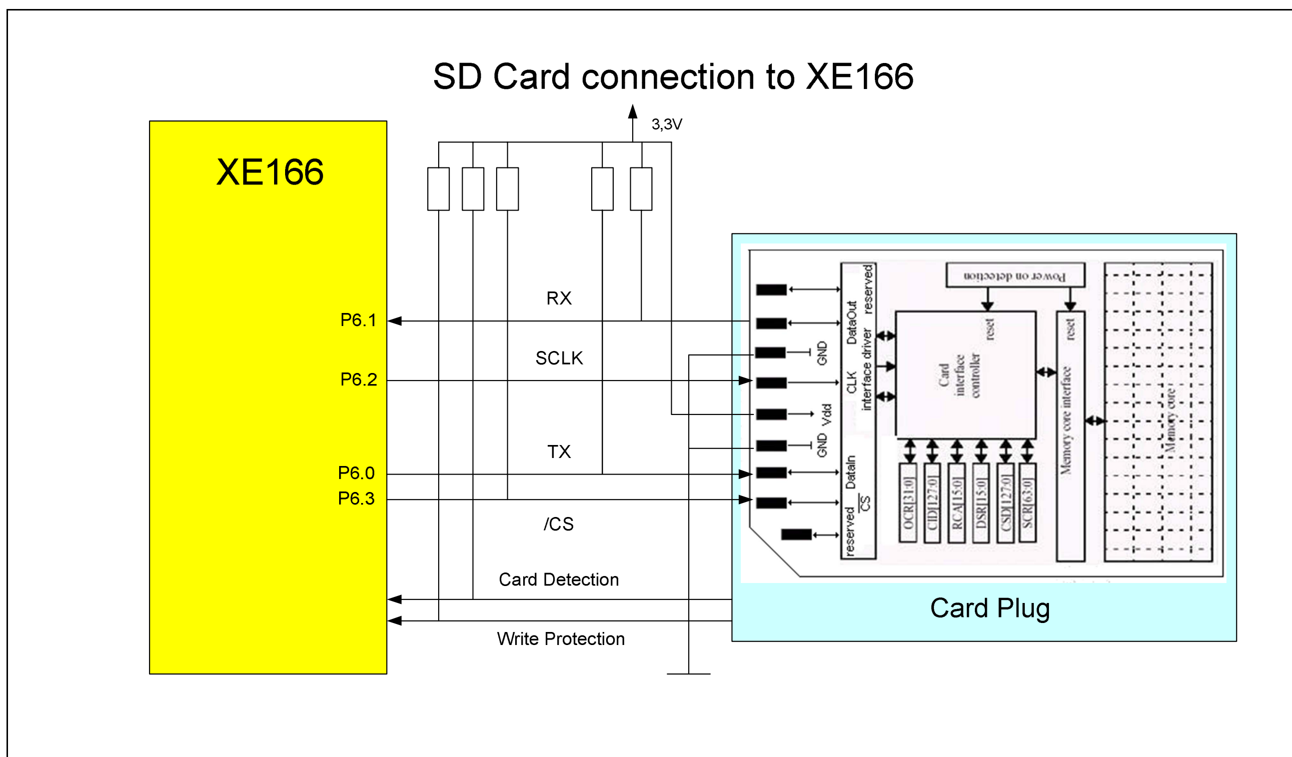
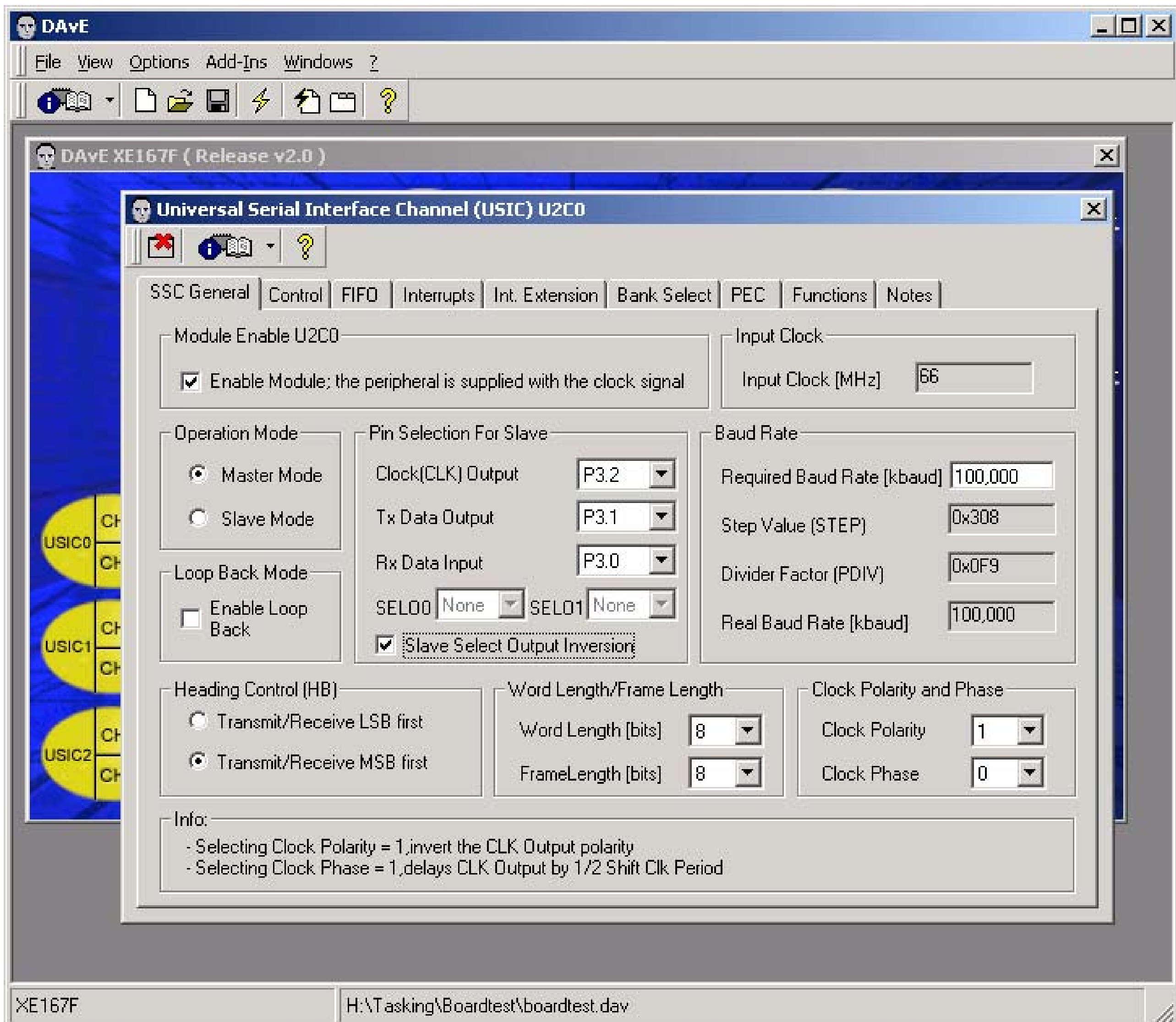


Figure 6 SD Card Connection for XE166

4.2 Software Setup

After power on reset, MMC/SDC enters its native operating mode. To switch to SPI mode, the procedure outlined in this section must be performed.



4.2.1 Power ON (Insertion)

After the supply voltage has reached 2.2 volts, wait for at least a millisecond then set DI and CS high, apply more than 74 pulses to SCLK, and then card will able to accept a native command.

4.2.2 Software Reset

Send a CMD0 with CS low to reset the card. The card samples CS signal when a CMD0 is detected. If the CS signal is low, the card enters SPI mode. Since the CMD0 must be sent as a native command, the CRC field must have a valid value. Once the card enters SPI mode the CRC check is disabled and the CRC value is negligible, so that command transmission routine can be written with the hardcoded CRC byte that is valid for only CMD0 and CMD8. When the CMD0 is accepted, the card will enter idle state and R1 response will be in In Idle State bit (0x01). The CRC can also be re-enabled with CMD59.

4.2.3 Initialization

In idle state, the card only accepts CMD0, CMD1 and CMD58. Any other commands will be rejected. When the card detects a CMD1, it starts initialization.

To poll end of the initialization, the host controller must repeat sending CMD1 and check the response. When the card is initialized successfully, In Idle State in the R1 response is cleared (0x00). The initialization process can take several hundred milliseconds (large cards tend to take longer), so this must be taken into consideration when determining the time out value.

After the initialization has finished, generic read/write commands will be able to be accepted. OCR and CSD can be read to confirm the card properties. The initial value of the block length can have a value larger than 512, so the block size should be re-initialized with CMD16 if necessary.

ACMD41 instead of CMD1 is recommended to initiate initialization for SDC. To support MMC and CMD, send ACMD41 first and when it is rejected, retry with CMD1, is ideal.

4.2.4 How to Support SDCard Ver2 and High Capacity Cards

After the card enters idle state with a CMD0, send a CMD8 with 0x1AA and correct CRC before initiating initialization.

When the CMD8 is rejected with an illegal command error, the card is SDC V1 or MMC.

When the CMD8 is accepted, R7 response (R1 + 32 bit return value) will be returned. The lower 12 bits in the return value 0x1AA mean that the card is SDC V2 and can work at a voltage range of between 2.7 to 3.6 volts. If this is not the case, the card cannot be used. Initiate initialization with ACMD41 with the HCS bit. After initialization is complete, read OCR and check the CCS[30] bit in the OCR.

When it is set, the addressing mode is block address and subsequent read/write command that described below is executed in block address instead of byte address. Further information can be taken out of SD-Card specification.

4.2.5 Communication with the SDCard

Every transaction with the card starts with sending a command to the card. The card sends a response to every command which it receives. The response from the card is called the response token. There are several different response tokens, depending on the command received. The most common are:

- R1 – 1Byte, response to all commands
- R2 – 2Byte, response to SEND_STATUS command
- R3 – 3Byte, response to SEND_OCR (operation condition register)
- Data response – 1Byte, response to every data block written to the card

The host has to receive the response token by sending dummy writes before it sends the next command or data block to the card. There are at least 4 bytes necessary to get an answer from the card, so wait another 4 bytes until sending the next command.

A command consists of 6 bytes (as shown in Figure 2). Byte 6 (CRC) is not necessary in SPI mode except for the CMD0 command. For other commands a blank byte should be sent.
CMD0 needs the CRC byte, but it's always the same command (argument is zero). The CRC is therefore 0x95.

Information held in bytes 2-5 depends on the command sent. Byte 1 contains the command. CMDXX corresponds to the HEX value 0x40 + XX. For example CMD13 = 0x40 + 0x20 = 0x4D = 01001101B.

The ACMDXX command consists of the CMD55 followed by a CMDXX. For example if the host wants to send the ACMD41 command it has to send CMD55 (0x95), receive the R1 response, send CMD41 (0x81) and receive the response again.

Byte 1				Byte 2-5		Byte 6	
7	6	5	0	31	0	7	0
0	1	Command		Command Argument		CRC	1

Figure 7 Structure of a Command

The rest of the commands are explained in the SDcard specification, so please refer to that document for more information about sending and reading data blocks.

4.2.6 Programmer Hints

4.2.6.1 Exact Control of Timing Issues

The most important point during programming is to keep the exact Notification of the SD-card specification. Timing issues in particular can cause heavy errors if they are not mentioned.

4.2.6.2 Extra 'waits'

For an unknown reason it was necessary to wait some time after sending the initial 74 clock-cycles. When the wait was not included, some cards could not be initialized, even when the SD-specification was followed exactly.

4.2.6.3 Problems with USIC-channel

Some problems were experienced with the USIC-Channel and its receive-interrupt. This problem was not solved, but by using Polling and the fact that the time is known when the right Data should be in the Receive-Buffer, this can be worked around.

4.2.6.4 Small Stack

Since the SD-card-communication is meant to be just an additional application of later Software, only a small user-stack is used. Unfortunately it is not big enough to hold all necessary information about the SD-card, the opened file, the data, and so on. As a work around, global variables can be used.

4.2.6.5 Naming the Files

The SD-specification is compatible with MMC-cards. It has not been tested but it might be possible that this code also works with MMC-cards. Therefore to distinguish between the two when writing a file to the card or reading from it, with an SD-card is inserted it is necessary to write S:\ and the filename (S stands for SD-card). The code also allows for M:\, to indicate an MMC-card, but this functionality is ***not*** tested.

5 Outstanding Tasks (to be done…)

The following are some of the outstanding tasks to be completed for working with the FAT-filesystem and the XE166-controller.

5.1 The Partitions

To get the cards to work it was necessary to implement some source code to get information about the partitions on a card. Some code must still be created that ensures that no file is written to an area which belongs to another partition. So the code currently only works for cards with only one partition.

5.2 Subdirectories

The handling of subdirectories is not tested, although there is some untested code in the FAT_LibPrivate.c. This code is to be used at the user's discretion..

The example-program hangs up, when you try to read out the data of a subdirectory using the function to read out a file. It is possible to decide between directory and file during the read-out of the directory-content. This still has to be implemented.

5.3 Long Filenames

Some source code for handling with long filenames is included to the example-code, but this is commented out because errors were discovered when working with it.

It will therefore be up to the user to work on this code if they wish to use long filenames. Documentation about it can be found in the Specification of FAT-Filesystems from Microsoft.

5.4 FAT32 / HCSD-Support

This example-code works for Cards up to 1GB (tested). For High-Capacity-SD-Cards it is necessary to change the init-process (please refer to SD-specification). It is also necessary to implement a FAT32 File-system that the HCSD-cards are formatted with. Some parts of the FAT-Library will work. Please refer to the FAT-specification for further details

http://www.infineon.com

Published by Infineon Technologies AG