Semiconductors

AN10406

Accessing MMC card using SPI on LPC2000

Rev. 02 — 28 November 2005 Application note

Document information

Info	Content
Keywords	MultiMediaCard (MMC), SPI, Microcontroller, MCU
Abstract	This document describes how to use the LPC2000 SPI interface to access MMC card

PHILIPS

Revision history

Rev	Date	Description
02	20051128	Error correction per customer request and optimizing the driver.
01	20051102	Initial version.

Contact information

For additional information, please visit: http://www.semiconductors.philips.com

For sales office addresses, please send an email to: sales.addresses@www.semiconductors.philips.com

1. Introduction

This application note describes how to use the SPI interface on the LPC2000 family MCU to access MultiMediaCard (MMC) card, provides sample software to access MMC card on the Keil's MCB214x board. It includes:

- General information on MMC protocol concept.
- Underlying SPI driver and design consideration for MMC communication.
- MMC commands, driver, and key APIs.
- Sample Software.

2. MMC Protocol Concept

2.1 MMC Bus Architecture

The MMC are the universal low cost, high speed (clock up to 20Mhz), data storage media. The bus architecture of MMC can be chosen in one of the two mode modes, MMC or SPI.

The MMC mode bus includes the following signals:

1. **CLK**: Host to card clock
2. **CMD**: Bi-directional Command/Response signal
3. **DATA0 to DATA3**: Four bi-directional data signal
4. **VDD, VSS**: Power and Ground signals

The SPI mode is compliant with the Serial Peripheral Interface (SPI) specification. Its bus architecture includes the following signals:

1. **CS**: Host to card Chip Select signal
2. **CLK**: Host to card clock signal
3. **MOSI**: (Master Out Slave In) Host to card single bit data signal
4. **MISO**: (Master In Slave Out) Card to host single bit data signal

Both MMC and SPI modes use the single master/multiple slave bus architecture to communicate with the end devices and/or media cards.

In this application note, only SPI mode is implemented to access the MMC media card.

2.2 MMC Bus Protocol

The SPI message consists of command, response, and data block token. The host initiates the communication. As described in the SPI specification, the host starts every bus transaction by asserting the **CS** signal low. All the commands and transmit data are sent to the media card on the **MOSI** line. All the command response and receiving data are received from the media card on the **MISO** line. **CLK** is used to synchronize the data transfer on the bus.

The communication between the host and the media card can be simply divided into two phases: card initialization phase and data access phase.

In initialization phase, the host uses a series of commands to reset the card, set the card to a known operational mode, set the block length for data access.

In data access phase, the host uses a series of commands to read and/or write data from the card after the initialization phase.

The complete command set can be found in the “The MultiMediaCard System System Specification version 3.1” by MMC Association Technical Committee.

3. SPI Driver and APIs for MMC communication

3.1 LPC2000 SPI Interface

All the MCUs in LPC2000 family have at least one SPI port.

Any of these SPI ports can be used for MMC communication because all these SPI ports can be configured as SPI master, and configured into one of the four modes by manipulating the CPOL and CPHA bits to alter the relationship between the data and clock phase.

The sample software has been tested on the LPC2148 MCU. On the LPC2148 MCU, there are one SPI port, or SPI0, and one Synchronous Serial Port (SSP), or SPI1, which can be configured for SPI communication. Due to the board configuration, the SPI interface used for MMC data communication is the embedded SSP port or SPI1.

The sample program below has been tested on Keil Software Inc.’s MCB214x board using their software development tool chain set, uVision IDE and Debugger, version 3. The single master, also referred as the SPI host, is the LPC2148 MCU, the slave device tested is a PQI’s 256MB MMC card on the SD/MMC slot on the MCB214x board.

3.2 SPI Initialization and General APIs

In order to configure SPI interface on the LPC2148 SSP port, the design consideration includes,

- GPIO setting. The SPI pins, CLK, CS, MOSI, MISO need to be configured through pin select and GPIO registers, PINSEL1, IODIR0, and IOSET0, before configuring the SPI interface.
- SPI clock pre-scale and clock rate. Based on the VPB clock (PCLK) setting in VPB Divider Control register (VPDIV), the clock pre-scale can be set through SSP Clock pre-scale Register (SSPCPSR), and the clock rate can be controlled in the SSP Control 0 Register (SSPCR0). The SPI clock rate in the sample test program is set to 4Mhz.
- SPI frame format and data size. The SPI format and data size can be configured through setting the proper clock polarity bit (CPOL) and clock phase bit (CPHA) and data size field (DSS) in the SSP control registers (SSPCR0). The data size is set to 8 bits/per frame, and both CPOL and CPHA bits are set to zero.
- SPI enable/disable. The SSP port should be disabled before the GPIO pin setting, clock pre-scale setting, frame format configuration, and enabled after all the configuration is done to ensure a clear start.

The SPI related APIs in the attached sample program include:

- void SPI_Init (void);

  Initializing SPI interface through configuring GPIO, VPBDIV, SSP port registers.

- void SPI_Send (unsigned char * data_pointer, unsigned int data_length);

  Sending a block of data based on the data pointer and the length of the data block.
- void SPI_Receive (unsigned char *data_pointer, unsigned int data_length);

  Receiving a block of data based on the data pointer and the length of the data block.
- unsigned char SPI_ReceiveByte(void);

  Receiving one byte of data, the return value of the API is the received data. This API is primarily used to obtain the command response at different phases.

4. Basic MMC commands

The commands mentioned below are referenced from MMC System Specification, as only minimum numbers of commands have been implemented in the sample program.

These commands are:

- CMD0 GO_IDLE_STATE, reset the card to idle state
- CMD1 SEND_OP_COND, ask the card in idle state to send their operation conditions contents in the response on the **MISO** line. Any negative response indicates the media card cannot be initialized correctly.
- CMD16 SET_BLOCKLEN, set the block length (in bytes) for all the following block commands, both read and write. In the sample program, the data length is set 512 bytes.
- CMD17 READ_SINGLE_BLOCK, read a block of data that its size is determined by the SET_BLOCKLEN command.
- CMD24 WRITE_BLOCK, write a block of data that its size is determined by the SET_BLOCKLEN command.

5. MMC Driver and key APIs

On the MMC host, MMC driver APIs use underlying SPI APIs to send a series of commands to initialize the MMC card, check the status on the command response, and then send a series of commands to read/write data to from the MMC card.

The MMC driver and some key APIs in the attached sample program include:

- unsigned char mmc_Init (void);

  This API is used to initialize the MMC card.

  Use SPI_Send() to send three commands CMD0, CMD1, CMD16, in sequence to initialize the MMC Card, then, use mmc_response(), described below, to get the response after each command from the card. If the response on each command is successful, the return value of mmc_init() indicates the status of the command response, zero is succeed and non-zero is failure.
- int mmc_write_block (unsigned int block_number);

  This API is used to send a block of data based on the block number to the MMC card.

  Use SPI_Send() to send CMD24 followed by the data and checksum, then use mmc_response() to get response after the command and the block of data, the return

value of mmc_write_block() indicates the status of the command and data response, zero is succeed and non-zero is failure.

- int mmc_read_block (unsigned int block_number);

  This API is used to receive a block of data based on the block number from the MMC card.

  Use SPI_Send() to send CMD17 to the MMC card first, check the response of the CMD17, if the response is successful, use SPI_ReceiveByte() repeatedly to read the data back from the MMC card. The Iteration of the SPI_ReceiveByte() is 512 (block length) + 2 (two-byte checksum). The return value of mmc_read_block() indicates the status of the command and data response, zero is succeed and non-zero is failure.

- Int mmc_response (unsigned char expected_response);

  Use SPI_ReceiveByte() to get the response from the **MISO** line, compare the received data with expected command response, if the responses match, mmc_response() will exit normally and return succeed value zero. If not within a certain period of time, mmc_response() will bail out and return non-zero failure value.

6. Sample Software

mmcmain.h

```
/*-------------------------------------------------------------------
 *      Name:    MMCMAIN.H
 *      Purpose: MMC Acess demo Definitions
 *      Version: V1.03
 *      Copyright (c) 2005 Philips Semiconductor. All rights reserved.
 *-------------------------------------------------------------------*/

#define CCLK            60000000    /* CPU Clock */

/* BLOCK number of the MMC card */
#define MAX_BLOCK_NUM   0x80

/* LED Definitions */
#define LED_MSK         0x00FF0000  /* P1.16..23 */
#define LED_RD          0x00010000  /* P1.16 */
#define LED_WR          0x00020000  /* P1.17 */
#define LED_CFG         0x00400000  /* P1.22 */
#define LED_DONE        0x00800000  /* P1.23 */
```

spi_mmc.h

```
/*-------------------------------------------------------------------
 *      Name:    SPI_MMC.H
 *      Purpose: SPI mode MMC card interface driver
 *      Version: V1.03
 *      Copyright (c) 2005 Philips Semiconductor. All rights reserved.
 *-------------------------------------------------------------------*/
#ifndef __SPI_MMC_H__
#define __SPI_MMC_H__

/* SPI select pin */
#define SPI_SEL       0x00100000

/* The SPI data is 8 bit long, the MMC use 48 bits, 6 bytes */
#define MMC_CMD_SIZE  6
```

```
/* The max MMC flash size is 256MB */
#define MMC_DATA_SIZE 512    /* 16-bit in size, 512 bytes */

#define MAX_TIMEOUT   0xFF

#define IDLE_STATE_TIMEOUT            1
#define OP_COND_TIMEOUT               2
#define SET_BLOCKLEN_TIMEOUT          3
#define WRITE_BLOCK_TIMEOUT           4
#define WRITE_BLOCK_FAIL              5
#define READ_BLOCK_TIMEOUT            6
#define READ_BLOCK_DATA_TOKEN_MISSING 7
#define DATA_TOKEN_TIMEOUT            8
#define SELECT_CARD_TIMEOUT           9
#define SET_RELATIVE_ADDR_TIMEOUT     10

void SPI_Init( void );
void SPI_Send( BYTE *Buf, DWORD Length );
void SPI_Receive( BYTE *Buf, DWORD Length );
BYTE SPI_ReceiveByte( void );

int mmc_init(void);
int mmc_response(BYTE response);
int mmc_read_block(WORD block_number);
int mmc_write_block(WORD block_number);
int mmc_wait_for_write_finish(void);

#endif  /* __SPI_MMC_H__ */
```

spi_mmc.c

```
/*-----------------------------------------------------------------------
 *      Name:    SPI_MMC.C
 *      Purpose: SPI and MMC command interface Module
 *      Version: V1.03
 *      Copyright (c) 2005 Philips Semiconductor. All rights reserved.
 *-----------------------------------------------------------------------*/
#include <LPC214X.H>                        /* LPC214x definitions */

#include "type.h"
#include "spi_mmc.h"

BYTE MMCWRData[MMC_DATA_SIZE];
BYTE MMCRDData[MMC_DATA_SIZE];
BYTE MMCCmd[MMC_CMD_SIZE];
BYTE MMCStatus = 0;

/*
 *  SPI and MMC commands related modules.
 *
 */
void SPI_Init( void )
{
  DWORD portConfig;
  BYTE i, Dummy;

  /* Configure PIN connect block */
  /* bit 32, 54, 76 are 0x10, bit 98 are 0x00
  port 0 bits 17, 18, 19, 20 are SSP port SCK1, MISO1, MOSI1,
  and SSEL1 set SSEL to GPIO pin that you will have the totoal
  freedom to set/reset the SPI chip-select pin */

  SSPCR1 = 0x00; /* SSP master (off) in normal mode */

  portConfig = PINSEL1;
  PINSEL1 = portConfig | 0x00A8;
  IODIR0 = SPI_SEL;/* SSEL is output */
  IOSET0 = SPI_SEL;/* set SSEL to high */

  /* Set PCLK 1/2 of CCLK */
  VPBDIV = 0x02;

  /* Set data to 8-bit, Frame format SPI, CPOL = 0, CPHA = 0,
  and SCR is 15 */
  SSPCR0 = 0x0707;

  /* SSPCPSR clock prescale register, master mode, minimum divisor
  is 0x02*/
  SSPCPSR = 0x2;

  /* Device select as master, SSP Enabled, normal operational mode */
  SSPCR1 = 0x02;

  for ( i = 0; i < 8; i++ )
  {
     Dummy = SSPDR;   /* clear the RxFIFO */
  }
  return;

}
```

```
/*
 * SPI Send a block of data based on the length
 */
void SPI_Send( BYTE *buf, DWORD Length )
{
  BYTE Dummy;

  if ( Length == 0 )
     return;
  while ( Length != 0 )
  {
     /* as long as TNF bit is set, TxFIFO is not full, I can write */
     while ( !(SSPSR & 0x02) );
     SSPDR = *buf;
     /* Wait until the Busy bit is cleared */
     while ( !(SSPSR & 0x04) );
     Dummy = SSPDR;        /* Flush the RxFIFO */
     Length--;
     buf++;
  }
  return;
}

/*
 * SPI receives a block of data based on the length
 */
void SPI_Receive( BYTE *buf, DWORD Length )
{
  DWORD i;

  for ( i = 0; i < Length; i++ )
  {
     *buf = SPI_ReceiveByte();
     buf++;
  }
  return;
}

/*
 * SPI Receive Byte, receive one byte only, return Data byte
 * used a lot to check the status.
 */
BYTE SPI_ReceiveByte( void )
{
  BYTE data;

  /* wrtie dummy byte out to generate clock, then read data from
  MISO */
  SSPDR = 0xFF;
  /* Wait until the Busy bit is cleared */
  while ( SSPSR & 0x10 );
  data = SSPDR;
  return ( data );
}

/************************* MMC Init ******************************/
/*
 * Initialises the MMC into SPI mode and sets block size(512), returns
 * 0 on success
 *
 */
int mmc_init()
{
```

```
DWORD i;

/* Generate a data pattern for write block */
for(i=0;i<MMC_DATA_SIZE;i++)
{
   MMCWRData[i] = i;
}

MMCStatus = 0;
IOSET0 = SPI_SEL; /* set SPI SSEL */

/* initialise the MMC card into SPI mode by sending 80 clks on */
/* Use MMCRDData as a temporary buffer for SPI_Send() */
for(i=0; i<10; i++)
{
   MMCRDData[i] = 0xFF;
}
SPI_Send( MMCRDData, 10 );

IOCLR0 = SPI_SEL; /* clear SPI SSEL */

/* send CMD0(RESET or GO_IDLE_STATE) command, all the arguments
are 0x00 for the reset command, precalculated checksum */
MMCCmd[0] = 0x40;
MMCCmd[1] = 0x00;
MMCCmd[2] = 0x00;
MMCCmd[3] = 0x00;
MMCCmd[4] = 0x00;
MMCCmd[5] = 0x95;
SPI_Send( MMCCmd, MMC_CMD_SIZE );

/* if = 1 then there was a timeout waiting for 0x01 from the MMC */
if( mmc_response(0x01) == 1 )
{
   MMCStatus = IDLE_STATE_TIMEOUT;
   IOSET0 = SPI_SEL; /* set SPI SSEL */
   return MMCStatus;
}

/* Send some dummy clocks after GO_IDLE_STATE */
IOSET0 = SPI_SEL; /* set SPI SSEL */
SPI_ReceiveByte();
IOCLR0 = SPI_SEL; /* clear SPI SSEL */

/* must keep sending command until zero response ia back. */
i = MAX_TIMEOUT;
do
{
   /* send mmc CMD1(SEND_OP_COND) to bring out of idle state */
   /* all the arguments are 0x00 for command one */
   MMCCmd[0] = 0x41;
   MMCCmd[1] = 0x00;
   MMCCmd[2] = 0x00;
   MMCCmd[3] = 0x00;
   MMCCmd[4] = 0x00;
   /* checksum is no longer required but we always send 0xFF */
   MMCCmd[5] = 0xFF;
   SPI_Send( MMCCmd, MMC_CMD_SIZE );
   i--;
} while ( (mmc_response(0x00) != 0) && (i>0) );

/* timeout waiting for 0x00 from the MMC */
if ( i == 0 )
```

```
  {
     MMCStatus = OP_COND_TIMEOUT;
     IOSET0 = SPI_SEL; /* set SPI SSEL */
     return MMCStatus;
  }

  /* Send some dummy clocks after SEND_OP_COND */
  IOSET0 = SPI_SEL; /* set SPI SSEL */
  SPI_ReceiveByte();
  IOCLR0 = SPI_SEL; /* clear SPI SSEL */

  /* send MMC CMD16(SET_BLOCKLEN) to set the block length */
  MMCCmd[0] = 0x50;
  MMCCmd[1] = 0x00;     /* 4 bytes from here is the block length */
                /* LSB is first */
                /* 00 00 00 10 set to 16 bytes */
                /* 00 00 02 00 set to 512 bytes */
  MMCCmd[2] = 0x00;
  /* high block length bits - 512 bytes */
  MMCCmd[3] = 0x02;
  /* low block length bits */
  MMCCmd[4] = 0x00;
  /* checksum is no longer required but we always send 0xFF */
  MMCCmd[5] = 0xFF;
  SPI_Send( MMCCmd, MMC_CMD_SIZE );

  if( (mmc_response(0x00))==1 )
  {
     MMCStatus = SET_BLOCKLEN_TIMEOUT;
     IOSET0 = SPI_SEL;  /* set SPI SSEL */
     return MMCStatus;
  }

  IOSET0 = SPI_SEL;   /* set SPI SSEL */
  SPI_ReceiveByte();
  return 0;
}

/************************** MMC Write Block ***************************/
/* write a block of data based on the length that has been set
 * in the SET_BLOCKLEN command.
 * Send the WRITE_SINGLE_BLOCK command out first, check the
 * R1 response, then send the data start token(bit 0 to 0) followed by
 * the block of data. The test program sets the block length to 512
 * bytes. When the data write finishs, the response should come back
 * as 0xX5 bit 3 to 0 as 0101B, then another non-zero value indicating
 * that MMC card is in idle state again.
 *
 */
int mmc_write_block(WORD block_number)
{
  WORD varl, varh;
  BYTE Status;

  IOCLR0 = SPI_SEL; /* clear SPI SSEL */

  /* block size has been set in mmc_init() */
  varl=((block_number&0x003F)<<9);
  varh=((block_number&0xFFC0)>>7);

  /* send mmc CMD24(WRITE_SINGLE_BLOCK) to write the data to MMC card */
  MMCCmd[0] = 0x58;
  /* high block address bits, varh HIGH and LOW */
```

```
  MMCCmd[1] = varh >> 0x08;
  MMCCmd[2] = varh & 0xFF;
  /* low block address bits, varl HIGH and LOW */
  MMCCmd[3] = varl >> 0x08;
  MMCCmd[4] = varl & 0xFF;
  /* checksum is no longer required but we always send 0xFF */
  MMCCmd[5] = 0xFF;
  SPI_Send(MMCCmd, MMC_CMD_SIZE );

  /* if mmc_response returns 1 then we failed to get a 0x00 response */
  if((mmc_response(0x00))==1)
  {
     MMCStatus = WRITE_BLOCK_TIMEOUT;
     IOSET0 = SPI_SEL;     /* set SPI SSEL */
     return MMCStatus;
  }

  /* Set bit 0 to 0 which indicates the beginning of the data block */
  MMCCmd[0] = 0xFE;
  SPI_Send( MMCCmd, 1 );

  /* send data, pattern as 0x00,0x01,0x02,0x03,0x04,0x05 ...*/
  SPI_Send( MMCWRData, MMC_DATA_SIZE );

  /* Send dummy checksum */
  /* when the last check sum is sent, the response should come back
  immediately. So, check the SPI FIFO MISO and make sure the status
  return 0xX5, the bit 3 through 0 should be 0x05 */
  MMCCmd[0] = 0xFF;
  MMCCmd[1] = 0xFF;
  SPI_Send( MMCCmd, 2 );

  Status = SPI_ReceiveByte();
  if ( (Status & 0x0F) != 0x05 )
  {
     MMCStatus = WRITE_BLOCK_FAIL;
     IOSET0 = SPI_SEL;     /* set SPI SSEL */
     return MMCStatus;
  }

  /* if the status is already zero, the write hasn't finished
  yet and card is busy */
  if(mmc_wait_for_write_finish()==1)
  {
     MMCStatus = WRITE_BLOCK_FAIL;
     IOSET0 = SPI_SEL;     /* set SPI SSEL */
     return MMCStatus;
  }

  IOSET0 = SPI_SEL;     /* set SPI SSEL */
  SPI_ReceiveByte();
  return 0;
}

/************************** MMC Read Block ***************************/
/*
 * Reads a 512 Byte block from the MMC
 * Send READ_SINGLE_BLOCK command first, wait for response come back
 * 0x00 followed by 0xFE. The call SPI_Receive() to read the data
 * block back followed by the checksum.
 *
 */
int mmc_read_block(WORD block_number)
```

```
{
  WORD Checksum;
  WORD varh,varl;

  IOCLR0 = SPI_SEL; /* clear SPI SSEL */

  varl=((block_number&0x003F)<<9);
  varh=((block_number&0xFFC0)>>7);

  /* send MMC CMD17(READ_SINGLE_BLOCK) to read the data from MMC card */
  MMCCmd[0] = 0x51;
  /* high block address bits, varh HIGH and LOW */
  MMCCmd[1] = varh >> 0x08;
  MMCCmd[2] = varh & 0xFF;
  /* low block address bits, varl HIGH and LOW */
  MMCCmd[3] = varl >> 0x08;
  MMCCmd[4] = varl & 0xFF;
  /* checksum is no longer required but we always send 0xFF */
  MMCCmd[5] = 0xFF;
  SPI_Send(MMCCmd, MMC_CMD_SIZE );

  /* if mmc_response returns 1 then we failed to get a 0x00 response */
  if((mmc_response(0x00))==1)
  {
     MMCStatus = READ_BLOCK_TIMEOUT;
     IOSET0 = SPI_SEL;     /* set SPI SSEL */
     return MMCStatus;
  }

  /* wait for data token */
  if((mmc_response(0xFE))==1)
  {
     MMCStatus = READ_BLOCK_DATA_TOKEN_MISSING;
     IOSET0 = SPI_SEL;
     return MMCStatus;
  }

  /* Get the block of data based on the length */
  SPI_Receive( MMCRDData, MMC_DATA_SIZE );

  /* CRC bytes that are not needed */
  Checksum = SPI_ReceiveByte();
  Checksum = Checksum << 0x08 | SPI_ReceiveByte();

  IOSET0 = SPI_SEL; /* set SPI SSEL */
  SPI_ReceiveByte();
  return 0;
}

/***************** MMC get response ******************/
/*
 * Repeatedly reads the MMC until we get the
 * response we want or timeout
 */
int mmc_response( unsigned char response)
{
  DWORD count = 0xFFF;

  while( (SPI_ReceiveByte() != response) && count )
  {
     count--;
  }
  if ( count == 0 )
```

```
    return 1;      /* Failure, loop was exited due to timeout */
  else
    return 0;      /* Normal, loop was exited before timeout */
}

/***************** MMC wait for write finish ******************/
/*
 * Repeatedly reads the MMC until we get a non-zero value (after
 * a zero value) indicating the write has finished and card is no
 * longer busy.
 *
 */
int mmc_wait_for_write_finish( void )
{
  DWORD count = 0xFFFF;    /* The delay is set to maximum considering
                              the longest data block length to handle */
  BYTE result = 0;

  while( (result == 0) && count )
  {
    result = SPI_ReceiveByte();
    count--;
  }
  if ( count == 0 )
    return 1;      /* Failure, loop was exited due to timeout */
  else
    return 0;      /* Normal, loop was exited before timeout */
}
```

mmcmain.c

```
/*-------------------------------------------------------------------
 *      Name:    MMCMAIN.C
 *      Purpose: MMC Card Access Demo
 *      Version: V1.03
 *      Copyright (c) 2005 Philips Semiconductor. All rights reserved.
 *-------------------------------------------------------------------*/
#include <LPC214X.H>                     /* LPC214x definitions */

#include "type.h"
#include "spi_mmc.h"
#include "mmcmain.h"

extern BYTE MMCWRData[MMC_DATA_SIZE];
extern BYTE MMCRDData[MMC_DATA_SIZE];

/* Main Program */
int main (void) {

   DWORD i, BlockNum = 0;

  PINSEL1 = 0x40004000;
  IODIR1  = LED_MSK;               /* LED's defined as Outputs */

  SPI_Init();              /* initialize SPI for MMC card */
  IOSET1 = LED_CFG;
  if ( mmc_init() != 0 )
  {
    IOSET0 = SPI_SEL;          /* set SSEL to high */
    while ( 1 );             /* Very bad happened */
  }
```

```
  /* write, read back, and compare the complete 64KB on the MMC
   * card each block is 512 bytes, the total is 512 * 128 */
  for ( BlockNum = 0; BlockNum < MAX_BLOCK_NUM; BlockNum++ )
  {
     IOCLR1 = LED_MSK;
     IOSET1 = LED_WR;
     if ( mmc_write_block(BlockNum) == 0 )
     {
        IOCLR1 = LED_MSK;
        IOSET1 = LED_RD;
        mmc_read_block(BlockNum);
     }
     else
     {
       IOSET0 = SPI_SEL;            /* set SSEL to high */
       while ( 1 );                 /* Very bad happened */
     }

     for ( i = 0; i < MMC_DATA_SIZE; i++ ) /* Validate */
     {
       if ( MMCRDData[i] != MMCWRData[i] )
       {
         IOSET0 = SPI_SEL;            /* set SSEL to high */
         while ( 1 );               /* Very bad happened */
       }
     }

     for ( i = 0; i < MMC_DATA_SIZE; i++ ) /* clear read buffer */
       MMCRDData[i] = 0x00;
  }

  IOCLR1 = LED_MSK;
  IOSET1 = LED_DONE;
  while (1);                                /* Loop forever */

}
```

7. References

- Philips LPC2142/2148 User Manual, Philips Semiconductors, May 2005.
- The MultiMediaCard System Specification, Version 3.1, MMC Association Technical Committee, June 2001.

8. Disclaimers

Life support — These products are not designed for use in life support appliances, devices, or systems where malfunction of these products can reasonably be expected to result in personal injury. Philips Semiconductors customers using or selling these products for use in such applications do so at their own risk and agree to fully indemnify Philips Semiconductors for any damages resulting from such application.

Right to make changes — Philips Semiconductors reserves the right to make changes in the products - including circuits, standard cells, and/or software - described or contained herein in order to improve design and/or performance. When the product is in full production (status 'Production'), relevant changes will be communicated via a Customer Product/Process Change Notification (CPCN). Philips Semiconductors assumes no responsibility or liability for the use of any of these products, conveys no licence or title under any patent, copyright, or mask work right to these products, and makes no representations or warranties that these products are free from patent, copyright, or mask work right infringement, unless otherwise specified.

Application information — Applications that are described herein for any of these products are for illustrative purposes only. Philips Semiconductors make no representation or warranty that such applications will be suitable for the specified use without further testing or modification.

9. Trademarks

Notice — All referenced brands, product names, service names and trademarks are the property of the respective owners.

10. Contents

Date of release:28 November 2005
Document number: AN10406_2

Published in The Netherlands