Interfacing a MultiMediaCard to the LH79520 System-On-Chip

INTRODUCTION

The MultiMediaCard (MMC) is a low-cost data storage media widely used in MP3 players, digital recorders, smart phones, PDAs, and pagers. It is common to find embedded MMC controllers in some high-end microcontrollers, but it is not necessary to use a hardware MMC controller to interface with the MMC card. The SPI peripheral in NXP's LH79520 microcontroller can easily handle this task.

This application note describes how to implement the interface between NXP's LH79520 System-on-Chip and a MultiMediaCard in both hardware and software. It will discuss using the LH79520 in MultiMediaCard applications, connecting the MultiMediaCard to the LH79520's built-in SPI controller, and how to read the FAT16 master boot block in the MultiMediaCard.

LH79520 Processor Bandwidth

The LH79520 is a 32-bit general-purpose microcontroller, using the ARM720T core in a 176-pin QFP package. The core can operate at up to 77.4 MHz and the bus can operate at up to 51.6 MHz. This is a System-on-Chip with many peripherals including MMU, CACHE, SSP, UART, SDRAM Controller, PWM, VIC, GPIO, and a 64 k-color LCD controller. Of these, the MultiMediaCard can best be connected through the SPI interface, which is supported by the SSP. The SPI controller in the LH79520 can operate at up to half of the bus clock speed, (approximately 25 Mbit/s) so this makes the SPI controller a good fit to drive the MultiMediaCard to achieve its maximum throughput of 20 Mbit/s.

Connecting the MultiMediaCard to the NXP LH79520 Microcontroller

The MultiMediaCard is based on an advanced 7-pin serial bus mode known as 'MultiMediaCard mode'. Most MultiMediaCards have a communication voltage from 2.0 V to 3.6 V, a memory access voltage of 2.7 V to 3.6 V, and the capacity can be anywhere from 4MB into the gigabyte range. The MultiMediaCard has two modes of operation, one called 'MultiMedia mode' and one called 'SPI mode'. A similar arrangement could be used for SD (Secure Digital) cards; however, this Application Note will only cover MMC cards in SPI mode; for MultiMedia mode, please refer to the MMC card specifications.

Table 1 shows the MMC pin assignments for SPI mode. Figure 1 shows a method of connection for SPI mode from the LH79520 to the MMC card. Note that pin 1 of the MMC card is tied to ground if there is only one MMC card in the system. If there are multiple MMC cards in the system, use a GPIO to control pin 1 of each card. When pin 1 is goes LOW, the corresponding MMC card is enabled.

A pull-up resistor on the DataIn pin and DataOut pin is necessary because the MMC card drives pins in 'Open Drain' mode. Caps between ground and power are important for noise reduction on the clock and data lines for the MMC.

Table 1. MMC Pin Assignment in SPI Mode

PIN	NAME	TYPE	SPI DESCRIPTION
1	nCS	Input	Chip Select (Active LOW)
2	DataIn	Input	Host-to-Card Commands and Data
3	VSS1	Power	Supply Voltage Ground
4	VDD	VCC	Supply Voltage
5	CLK	Input	Clock
6	VSS2	Power	Supply Voltage Ground
7	DataOut	Output	Card-to-Host Data and Status

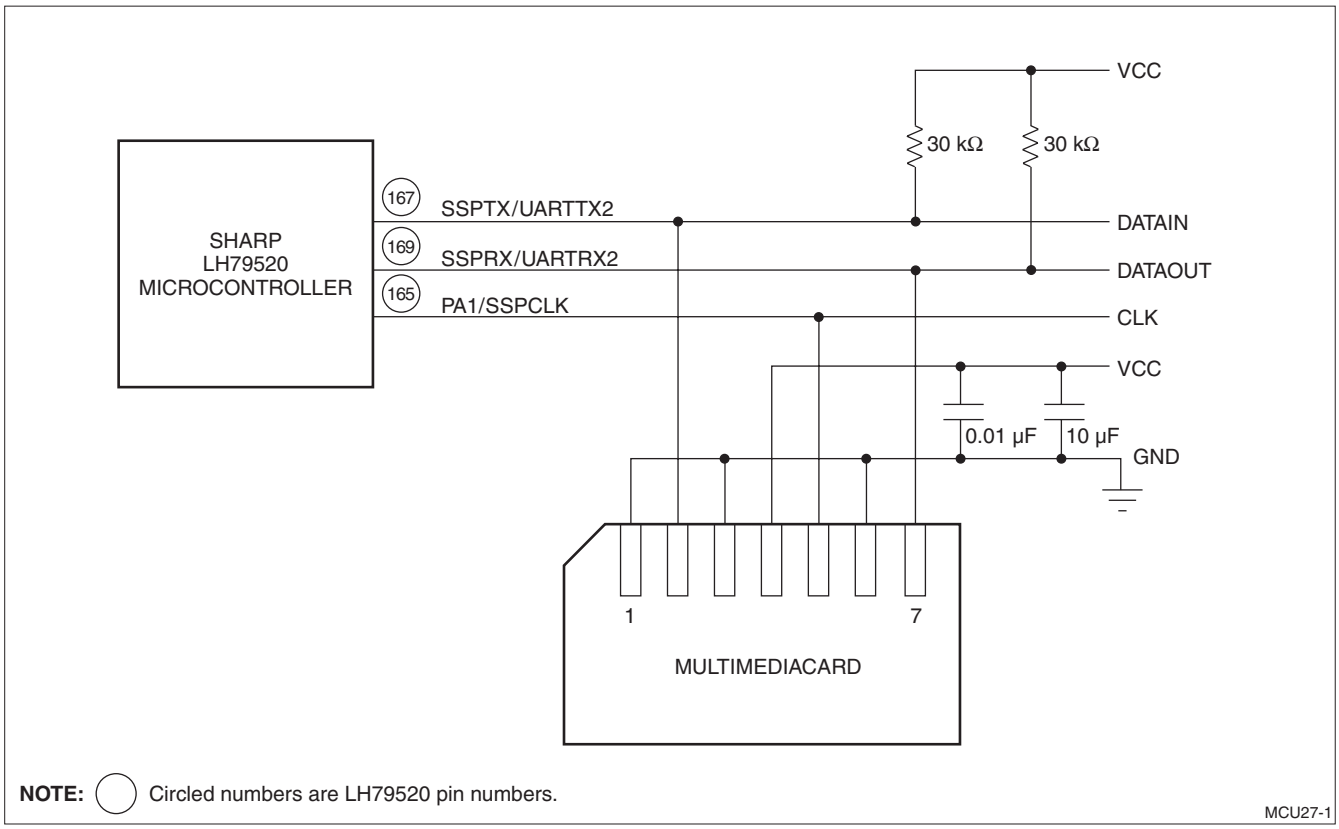

Figure 1. LH79520-to-MultiMediaCard Connection

SPI Commands

Communications between the microcontroller and the MMC are initiated by different commands sent from the microcontroller to the MMC. The most common of these commands are listed in Table 2; for the complete command set, refer to the MMC specifications.

Table 2. SPI Commands

CMD INDEX	ARGUMENT	RESPONSE	ABBREVIATION	COMMAND DESCRIPTION
CMD0	None	R1	GO_IDLE_STATE	Resets the MultiMediaCard
CMD1	None	R1	SEND_OP_COND	Activates the card Initialization process
CMD13	None	R2	SEND_STATUS	Asks the selected card to send its status register
CMD16	[31:0]block length	R1	SET_BLOCKLEN	Selects a block length (in bytes) for all following block commands (read and write).
CMD17	[31:0]data address	R1	READ_SINGLE_BLOCK	Reads a block of size selected by the SET_BLOCKLEN command
CMD24	[31:0]data address	R1	WRITE_BLOCK	Writes a block of the size selected by the SET_BLOCKLEN command
CMD32	[31:0]data address	R1	TAG_SECTOR_START	Sets the address of the first sector of the erase group
CMD33	[31:0]data address	R1	TAG_SECTOR_END	Sets the address of the last sector in a continuous range within the selected erase group, or the address of a single sector to be selected for erase.
CMD34	[31:0]data address	R1	UNTAG_SECTOR	Removes one previously selected sector from the erase selection
CMD38	[31:0]don't care	R1b	ERASE	Erases all previously selected sectors
CMD59	[31:1]don't care [0:0]CRC option	R1	CRC_ON_OFF	Turns the CRC option on or off. A '1' in the CRC option bit will turn the option on. A '0' will turn it off.

COMMAND TRANSMISSION

All commands are 6 bytes long and are transmitted MSB first.

Table 3. Command Transmissions

BYTE 1	7	6	5	4	3	2	1	0
FIELD	0	1	Command					

BYTES 2 - 5	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
FIELD	Argument															
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
FIELD	Argument															

BYTE 6	7	6	5	4	3	2	1	0
FIELD	CRC						1	x

CRC CALCULATION

The CRC bit calculation is performed:

7-bit CRC Calculation: $G(x) = x^7 + x^3 + 1$

$M(x) = (\text{start bit}) \times x^{39} + (\text{second bit}) \times x^{38} + \ldots + (\text{last bit before CRC}) \times x^0$

$CRC[6\ldots0] = \text{Remainder}[(M(x) \times x^7)/G(x)]$

RESPONSE FORMAT R1

This response token is sent by the card after every command with the exception of SEND_STATUS commands. It is 1 byte long; the MSB is always set to zero and the other bits are error indications. A '1' signals an error.

The structure of the R1 format is given in Table 4 and Table 5.

RESPONSE FORMAT R1B

This response token is identical to the R1 format with the addition of the optional BUSY signal. The Card holds the DataIn line LOW to signal BUSY; this can last for any period until the Card has finished processing the current transaction. Once the Card releases the line, it is ready for the next command.

Table 4. R1 Format Byte Structure

BIT	7	6	5	4	3	2	1	0
FIELD	0	Parameter Error	Address Error	Erase Seq Error	Com CRC Error	Illegal Command	Erase Reset	In Idle State

Table 5. R1 Format Byte Definitions

BIT	NAME	DESCRIPTION
7	0	Fixed to '0'
6	Parameter Error	The command's argument (e.g. address, block length) was out of the allowed range for this card
5	Address Error	A misaligned address, which did not match the block length, was used in the command
4	Erase Seq Error	An error in the sequence of erase commands occurred
3	Com CRC Error	The CRC check of the last command failed
2	Illegal Command	An illegal command code was detected
1	Erase Reset	An erase sequence was cleared before executing because an out of erase sequence command was received
0	In Idle State	The card is in idle state and running initializing process

RESPONSE FORMAT R2

This 2-byte-long, response token is sent by the card as a response to the SEND_STATUS command. The format of the R2 status is given in Table 6 and Table 7.

Table 6. Response Format R2 Bits

BITS	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
FIELD	0	Parameter Error	Address Error	Erase Seq Error	Com CRC Error	Illegal Command	Erase Reset	In Idle State	Out of Range	Erase Param	WP Violation	Card ECC Failed	CC Error	Error	WP Erase Skip	0

Table 7. Response R2 Format Definitions

BIT	NAME	DEFINITION
15	0	Fixed to ‘0’
14	Parameter Error	The command’s argument (e.g. address, block length) was out of the allowed range for this card
13	Address Error	A misaligned address, which did not match the block length was used in the command
12	Erase Seq Error	An error in the sequence of erase commands occurred
11	Com CRC Error	The CRC check of the last command failed
10	Illegal Command	An illegal command code was detected
9	Erase Reset	An erase sequence was cleared before executing because an out of erase sequence command was received
8	In Idle State	The card is in idle state and running initializing process
7	Out of Range	
6	Erase Param	An invalid selection, sectors or groups, for erase
5	WP Violation	The command tried to write a write protected block
4	Card ECC Failed	Card internal ECC was applied but failed to the corrected data
3	CC Error	Internal card controller error
2	Error	A general or an unknown error occurred during the operation
1	WP Erase Skip	Only partial address space was erased due to existing WP blocks
0	0	Fixed to ‘0’

DATA RESPONSE

Every data block written to the card will be acknowledged by a data response token. It is one byte long and has a format as seen in Table 8.

Table 8. Data Response Byte Structure

BIT	7	6	5	4	3	2	1	0
FIELD	0	0	0	0	Status			1

The status bits may be one of two states:

‘010’ = Data accepted

‘101’ = Data rejected due to a CRC error

DATA TOKENS

Read and write commands have data transfers associated with them. Data is being transmitted or received via data tokens. All data bytes are transmitted MSB first.

Data tokens are 4 to 515 bytes long and have a format as seen in Table 9.

Data Token bytes 2 to 513 can be any data block length, since their payload is User Data.

The last two bytes of the Data Token are a 16-bit CRC.

Table 9. Data Token Start Byte (Byte 1) Structure

BIT	7	6	5	4	3	2	1	0
FIELD	1	1	1	1	1	1	1	0

DATA ERROR TOKEN

If a read operation fails and the card can not provide the required data it will send a data error token, instead. This token is one byte long and has a format as seen in Table 10.

Table 10. Data Error Token Structure

BIT	7	6	5	4	3	2	1	0
FIELD	0	0	0	0	Out_of_Range	Card_ECC_Failed	CC_Error	Error

SPI Protocol

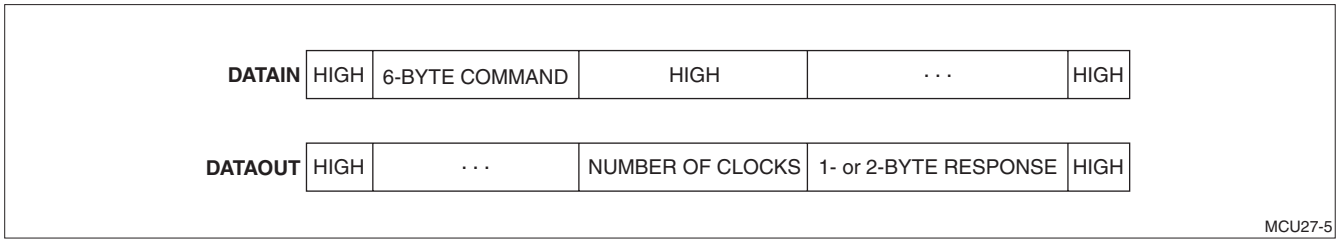

Figure 2. Host Command to Card Response — Card is Ready

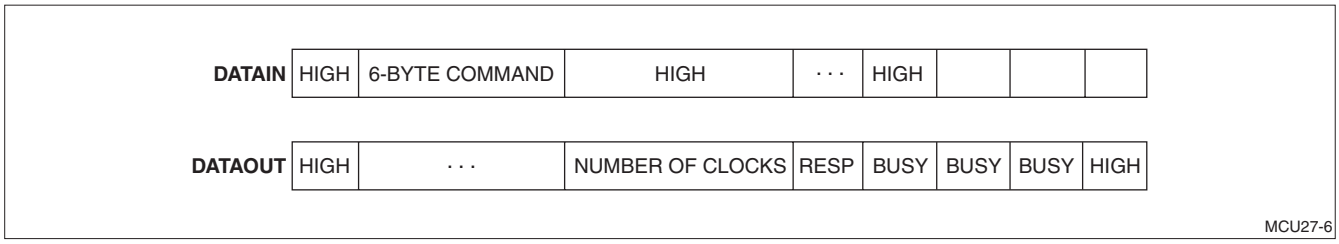

Figure 3. Host Command to Card Response — Card is Busy

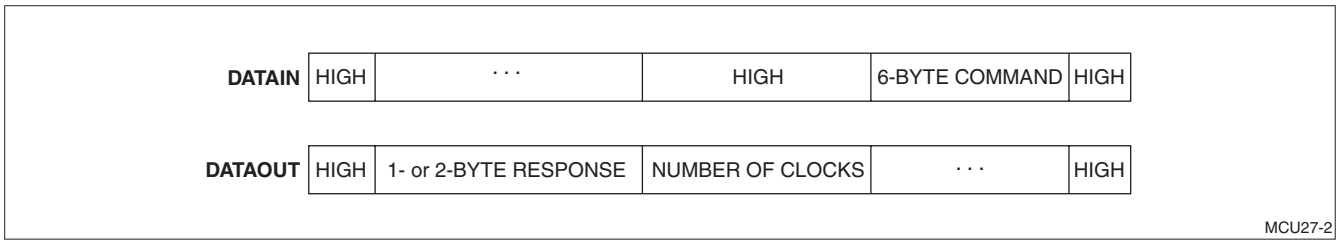

Figure 4. Card Response to Host Command

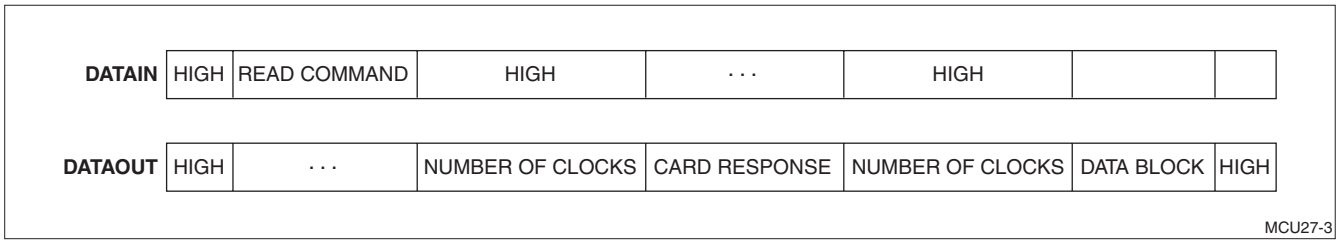

Figure 5. Data Read

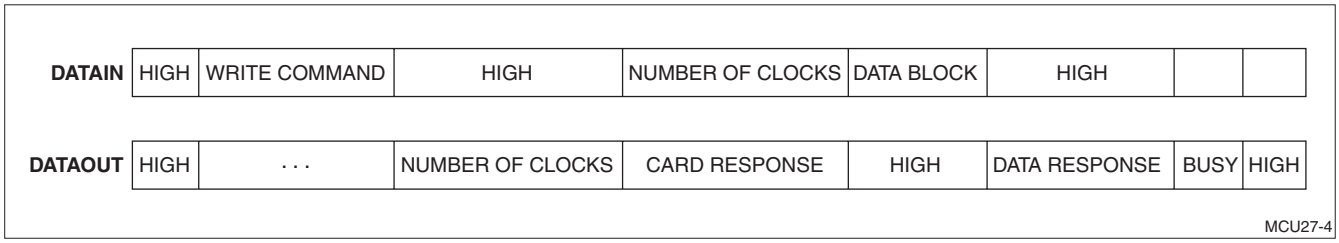

Figure 6. Data Write

Software Implementation

A source code listing for the MMC function is provided with this application note. See 'Code Listing'.

NXP SPI DRIVER

NXP source code for LH79520 microcontroller drivers (SPI, UART, LCD, and so on) can be downloaded from the NXP Semiconductors web page, at www.sharpsma.com. The tag for the source code is called the ABL BlueStreak Software Library. Within this Library, you can download the full source code for all NXP microcontroller drivers. The MMC interface source code is based on the LH79520 drivers.

The LH79520 Synchronous Serial Port (SSP) peripheral supports devices utilizing Motorola SPI, National Semiconductor Mircrowire or Texas Instruments's Synchronous Serial interfaces.

The SSP performs parallel-to-serial conversion on data written to an internal Transmit FIFO, then transmits the data, in serial fashion, to an external slave peripheral. The SSP also receives serial data from an external slave peripheral, performs a serial-to-parallel conversion on the received data, and buffers the received data to an internal Receive FIFO. Both FIFOs are 16 bits wide × 8 storage locations deep. Data frame sizes may be programmed to be from 4 to 16 bits in length.

The LH79520 DMAC (DMA controller) can be programmed to transfer data to and from the on-chip SSP. For more information, you may wish to download and refer to the LH79520 User's Guide, available on www.sharpsma.com.

The driver for the LH79520 SSP is named lh79520_ssp_driver.c. The driver for the MMC interface is named lh79520_mmc_driver.c, see the 'Code Listing' section.

CLOCK CONTROL

The SPI bus clock signal can be used by the SPI host to set the cards to energy saving mode or to control data flow (to avoid under-run or over-run conditions) on the bus. The host is allowed to change the clock frequency or stop it altogether.

There are a few restrictions the SPI host must follow:

- The bus frequency can be changed at any time, but only up to the maximum data transfer frequency, defined by the MultiMediaCards.
- It is an obvious requirement that the clock must be running for the MultiMediaCard to output data or response tokens. After the last SPI bus transaction, the host is required to provide 8 clock cycles for the card to complete the operation before shutting down the clock. During this 8-clock period, the state of the CS signal is irrelevant. It can be asserted or de-asserted.

SPI BUS TRANSACTIONS

Here is a list of the various SPI bus transactions:

- A command/response sequence. Eight clocks must be output after the card response end bit. The CS signal can be asserted or de-asserted during these 8 clocks.
- A read data transaction. Eight clocks must be output after the end bit of the last data block.
- A write data transaction. Eight clocks must be output after the end bit of the last data block.
- A write data transaction. Eight clocks must be output after the CRC status token.
- The host is allowed to stop the clock of a BUSY card. The MultiMediaCard will complete the programming operation regardless of the host clock. However, the host must provide a clock edge for the card to turn off its BUSY signal. Without a clock edge, the MultiMediaCard (unless previously disconnected by de-asserting the CS signal) will force the DataOut line LOW and hold it there.

MODE SELECTION

The MultiMediaCard's SPI mode is the mode used for this Application Note. All transactions described in this Application Note are based on the SPI mode.

The MultiMediaCard wakes up in the MultiMediaCard mode. It will enter SPI mode if the CS signal (pin 1 of the MMC) is asserted LOW during the reception of the Reset command (CMD0). If the card is in MultiMediaCard mode, it will not respond to SPI-based commands. If SPI mode is requested, the card will switch to SPI mode and respond with the SPI mode R1 response.

To return to the MultiMediaCard mode, power cycle the card. In SPI mode, the MultiMediaCard protocol state machine is not observed. MultiMediaCard commands supported in SPI mode are always available.

Since the card defaults to MultiMediaCard mode after a power cycle, Pin 1 (CS) must be pulled LOW and CMD0 (followed by a valid CRC byte) must be sent on the CMD (DataIn, Pin 2) line for the card to enter SPI mode.

In SPI mode, CRC checking is disabled by default. However, since the card always powers up in MultiMediaCard mode, CMD0 must be followed by a valid CRC byte (even though the command is sent using the SPI structure). Once the card enters SPI mode, CRCs are disabled by default.

CMD0 is a static command and always generates the same 7-bit CRC of 4Ah. Adding the '1' end bit (bit 0) to the CRC creates a CRC byte of 95h. The following hexadecimal sequence can be used to send CMD0 in all situations for SPI mode, since the CRC byte (although required) is ignored once in SPI mode. The entire CMD0 appears as: 40 00 00 00 00 95 (hexadecimal).

This CMD0 command (0x40 0x00 0x00 0x00 0x00 0x95) is the same command to switch the MMC card from MultiMediaCard mode to SPI mode. After this command is sent, CRC checking is disabled by default unless you want to enable it. When CRC checking is off, the last byte in a 6-byte command is ignored for read/write commands.

COMMAND AND RESPONSE

In the MMC command format, a command is comprised of 6 bytes and is sent MSB first. Once the SPI mode is set for 8-bit data width, 6-byte commands can be sent continuously. See the MMC_send_cmd ( ) function in the Code Listing.

Command responses may get a little tricky. The starting bit of the response may not align with the first clock of the byte. The starting bit of the response may happen anywhere in the clock stream, depending on the speed of the MMC and the clock. So there is a need for manual alignment in software. See the MMC_get_response ( ) function in the Code Listing.

RESET SEQUENCE

The initialization command is described in the following sequence:

1. Send 80 clocks to start bus communication
2. Assert nCS LOW
3. Send CMD0
4. Send 8 clocks for delay
5. Wait for a valid response
6. If there is no response, back to step 4
7. Send 8 clocks of delay
8. Send CMD1
9. Send 8 clocks of delay
10. Wait for valid response
11. Send 8 clocks of delay
12. Repeat from step 9 until the response shows READY.

It will take a large number of cycles for CMD1 to finish its sequence. After every power cycle, the MMC will be in Idle state (not active), the Idle bit in its response will be 1 if using CMD13 (SEND_STATUS) to check the status. Once the CMD1 process is finished, the Idle bit in the response is cleared. Only after MMC is fully up from Idle mode to Active, can it be read and written.

See the MMC_init ( ) function in the Code Listing.

DATA READ

The SPI mode supports single block read operations only. Upon reception of a valid Read command, the card will respond with a Response token followed by a Data token in the length defined by a previous SET_BLOCK_LENGTH command. The start address can be any byte address in the valid address range of the card. Every block however, must be contained in a single physical card sector. After the Data Read command is sent from microcontroller to the card, the microcontroller will need to monitor the data stream input and wait for Data Token 0xFE. Since the response start bit 0 can happen any time in the clock stream, it's necessary to use software to align the bytes being read.

See the MMC_start_sector_read ( ) function in the Code Listing.

DATA WRITE

Data Write operations are similar to Data Read. In SPI mode, the MMC supports single block writes only. Upon reception of a valid Write command, the card will respond with a Response token and wait for a data block to be sent from the host. The only valid block length, however, is 512 bytes. After a data block is received, the card will respond with a Data-response token and if the data block is received with no errors, it will be programmed.

The microcontroller must first send the Write command, followed by the bytes to be written. After all the bytes have been sent, the microcontroller waits for the response. Based on the response received, the microcontroller can check whether there is any error in the response. After the response is sent back from the card, the card will set DataOut LOW because it will take time to do the write.

See the MMC_start_sector_write ( ) and mmc_write_data( ) function in the Code Listing.

DATA ERASE

Data erase follows a similar sequence to Data Read and Data Write. See mmc_erase_sector ( ) function in the Code Listing.

READING FAT16 FILE SYSTEM MASTER BOOT BLOCKS

Although the MultiMediaCard memory space is byte-addressable with addresses ranging from 0 to the last byte, it is not a simple byte array but rather it is divided into several structures.

Memory bytes are grouped into 512-byte blocks called sectors. Every block can be individually read, written, and erased.

Sectors are grouped into Erase groups of 16 or 32 sectors depending on card size. Any combination of sectors within one group or any combination of Erase groups can be erased in a single Erase command. A Write command implicitly erases the memory before writing new data into it. An explicit Erase command can be used for pre-erasing memory to speed up the next Write operation.

The FAT16 file system is commonly used on PCs, and it's easy to find a card reader to format the Multi-MediaCard. After the MMC is formatted in FAT16 format, byte 0 to byte 0x200 will be used for the FAT16 format. This 512 bytes in sector 0 is also called the Master Boot Record (MBR) of the card, and will be in this format as shown in Table 13.

Inside the MBR, each partition entry is defined as shown in Table 14.

Table 13. FAT16 Master Boot Record

OFFSET	SIZE	DESCRIPTION
000h	446 bytes	Executable code (Boots Computer)
1BEh	16 bytes	1st Partition Entry (See Table 14)
1CEh	16 bytes	2nd Partition Entry
1DEh	16 bytes	3rd Partition Entry
1EEh	16 bytes	4th Partition Entry
1FEh	2 bytes	Executable Marker (55h AAh)

Table 14. FAT16 Partition Entries

OFFSET	SIZE	DESCRIPTION
00h		Current State of Partition, 1 byte (00h = Inactive, 80h = Active)
01h	1 byte	Beginning of Partition – Head
02h	2 bytes	Beginning of Partition – Cylinder/Sector
04h	1 byte	Type of Partition (See Table 15)
05h	1 byte	End of Partition – Head
06h	1 word	End of Partition – Cylinder/Sector
08h	1 double word	Number of Sectors Between the MBR and First Sector in the Partition
0Ch	1 double word	Number of Sectors in the Partition

Partition Type Table

Software should first read the Master Boot Record (MBR). The address for this command is 0x0. If you can read this block properly, the rest the FAT16 file system will follow easily.

Table 15. Partition Types

VALUE	DESCRIPTION
00h	Unknown or Nothing
01h	12 bit FAT
04h	16 bit FAT (Partition Smaller than 32MB)
05h	Extended MS-DOS Partition
06h	16 bit FAT (Partition Larger than 32MB)
0Bh	32 bit FAT (Partition Up to 2048GB)
0Ch	Same as 0Bh, but uses LBA1 13h Extensions
0Eh	Same as 06h, but uses LBA1 13h Extensions
0Fh	Same as 05h, but uses LBA1 13h Extensions

CONCLUSION

Using the LH79520's SPI interface is a practical way to read and write to a MultiMediaCard. This Application Note provides you with an operational basis for using an MMC in a design with the LH79520. For a detailed software implementation, see the Code Listing.

CODE LISTING

```
* $Workfile:   $
* $Revision:   $
* $Author:    JUN LI $
* $Date:       $
*
* Project: MMC card interface driver
*
* Description:
*  This driver is based on the compact flash driver.
*  This driver supports a MMC card..
*
* Notes:
*  The LH79520_MMC_driver header file is included in the this file, as
*  some MMC specific data is defined there.
*
* Revision History:
* $Log:   $
*
*
* NXP SEMICONDUCTORS MAKES NO REPRESENTATION
* OR WARRANTIES WITH RESPECT TO THE PERFORMANCE OF THIS SOFTWARE,
* AND SPECIFICALLY DISCLAIMS ANY RESPONSIBILITY FOR ANY DAMAGES,
* SPECIAL OR CONSEQUENTIAL, CONNECTED WITH THE USE OF THIS SOFTWARE.
*
* NXP SEMICONDUCTORS PROVIDES THIS SOFTWARE SOLELY
* FOR THE PURPOSE OF SOFTWARE DEVELOPMENT INCORPORATING THE USE OF A
* NXP SEMICONDUCTORS OR MICROCONTROLLER PRODUCT. USE OF THIS SOURCE
* FILE IMPLIES ACCEPTANCE OF THESE CONDITIONS.
*
* COPYRIGHT (C) 2007 NXP SEMICONDUCTORS
*****************************************************************************/
#include "abl_types.h"
#include "lh79520_ssp_driver.h"
#include "LH79520_lpd_mmc_driver.h"

static void spi_delay_bytes(INT_32 n_8_clks);
static UNS_16 get_crc16(CHAR * uc_data, INT_32 data_length);
static CHAR get_crc7(CHAR * cmd_bytes);
static void MMC_send_cmd(UNS_8 cmd, UNS_32 arg);
static INT_32 MMC_get_response(UNS_8 type);
static INT_32 ssp_transceive_word(INT_32 data);
```

```
//Local variables
STATIC INT_32 dev_ssp = 0;

INT_32 mmc_data_addr = 0;//Current data address for MMC operation
INT_32 block_len = 512;

UNS_8 mmc_dat_pos = 0;//data read write first byte starting 0 position
UNS_8 mmc_dat_old = 0;//data read write old byte holding alignment

/***********************************************************************
 * Function: mmc_init
 *
 * Purpose:  Initialize the MMC interface and return the card detection
 *           status.
 *
 * Processing:  The pointers used in this driver are initialized.
 *
 * Parameters:  None
 *
 * Outputs:  None
 *
 * Returns:  '1' if a MMC card has been detected, '0' otherwise.
 *
 * Notes:  None
 *
 ***********************************************************************/
INT_32 mmc_init (void)
{
    INT_32 ready, response;

       // Do SSP device initialization since SSP is used for
       // MMC interface
    // Open SSP
    if ((dev_ssp = ssp_open(SSP,0)) == 0x0)
    {
        // Error opening the device
        return 0;
    }

    // Set SSP frame format - Motorola SPI format
    ssp_ioctl(dev_ssp, SSP_SET_FRAME_FORMAT, SSP_MODE_MOTOROLA);

    // Set SSP data size
    ssp_ioctl(dev_ssp, SSP_SET_DATA_SIZE, 8);

    // Set SSP speed in us
    ssp_ioctl(dev_ssp, SSP_SET_SPEED, 20000000);
```

```
// Set SSPFRM pin logic level - We don't care SSPFRM
ssp_ioctl(dev_ssp, SSP_SET_SSPFRM_PIN, SSPFRM_AUTO);

// Set SCLK polarity as normal polarity - rising edge trigger data
ssp_ioctl(dev_ssp, SSP_SET_SCLK_POLARITY, 0);

// Set SCLK phase as normal - SSPFRM control, we don't card
ssp_ioctl(dev_ssp, SSP_SET_SCLK_PHASE, 0);

// Enable SSP
ssp_ioctl(dev_ssp, SSP_ENABLE, 1);

   spi_delay_bytes(10);// Delay 80 colcks

   // MMC is initializaed for use
ready = mmc_is_card_inserted ();

if(ready == 0)
{
   return 0;
}

   //Force the card to idle state
   response = 0xff;

   while(response != R1_IN_IDLE_STATE)
   {
          MMC_send_cmd(CMD_GO_IDLE_STATE, 0);
          response = MMC_get_response(RESP_R1);
   }

   //Turn off CRC
   MMC_send_cmd(CMD_CRC_ONOFF, 0);
   response = MMC_get_response(RESP_R1);

   //Do MMC init process
   response = 0xff;
   while(response != 0x0)
   {
          MMC_send_cmd(CMD_SEND_OP_COND, 0);
          response = MMC_get_response(RESP_R1);
   }

   // MMC is initializaed for use
ready = mmc_is_card_ready ();

   if(ready == 1)
```

```
	{
		// Send Block length as 512 for further operation
		MMC_send_cmd(CMD_SET_BLOCKLEN, 512);
		response = MMC_get_response(RESP_R1);
		if(response != 0)
			ready = 0;
	}

    return ready;
}

/*********************************************************************
 * Function: mmc_set_sector
 *
 * Purpose:  Set the cylinder, head, and sector for the next operation
 *           (using the absolute sector number).
 *
 * Processing:  The sector passed from the caller update the CHS
 *              device pointers that will be used for the next
 *              operation.
 *
 * Parameters:  sectorno : Sector number
 *
 * Outputs:  None
 *
 * Returns:  Nothing
 *
 * Notes:  The convention is Cylinder/Head/Sector (CHS). The function
 *         will convert the CHS values to a value that works with the
 *         MMC card.
 *
 *********************************************************************/
void mmc_set_sector (UNS_32 sectorno)
{
	mmc_data_addr = sectorno * 512;
}

/*********************************************************************
 * Function: mmc_start_sector_read
 *
 * Purpose:  Starts the read of a sector.
 *
 * Processing:  Set the sector size to '1' and issue the sector read command.
 *
 * Parameters:  None
 *
 * Outputs:  None
 *
```

```
 * Returns:  Nothing
 *
 * Notes:  None
 *
 ***************************************************************************/
void mmc_start_sector_read (void)
{
       INT_32 response;
       INT_8 i;
       UNS_8 tmp, tmp1, pos;

       // Send command to initialize the MMC card
       MMC_send_cmd(CMD_READ_BLOCK, mmc_data_addr);
       response = MMC_get_response(RESP_R1);

       if(response != 0)
       {
              return;
       }
       else
       {
              tmp = tmp1 = 0xff;
              //Wait until error token or data token received
              while(tmp1 == 0xff)
              {
                     tmp1 = ssp_transceive_word(tmp);
                     //tmp1 = ssp_ioctl(dev_ssp, SSP_TX_RX_WORD, tmp);
              }

       }

       pos = 0x80;
       //Now find the starting bit of data block
       for(i=0;i<8;i++)
       {
              if((tmp1 & (pos>>i)) == 0)
              {
                     mmc_dat_pos = i+1;
                     //if(mmc_dat_pos>7) mmc_dat_pos=0;
                     break;
              }
       }

       mmc_dat_old = tmp1;

}

/***************************************************************************
```

```
 * Function: mmc_read_data
 *
 * Purpose:  Read a block of data from the MMC card.
 *
 * Processing:  Copy a block of data of from the MMC card buffer to the
 *              destination address.
 *
 * Parameters:
 *          data  : Pointer to where to put read data from the MMC card
 *          bytes : Number of bytes to read
 *
 * Outputs:  The data pointed to by data will be updated.
 *
 * Returns:  The number of bytes read from the card.
 *
 * Notes:  This function will read out 512 bytes fixed
 *
 ********************************************************************/
INT_32 mmc_read_data (void *data, INT_32 bytes)
{
    INT_32 i;
       UNS_8 tmp, tmp1;
    UNS_8 * _data = (UNS_8 *)data;

       tmp = tmp1 = 0xff;
       for(i=0;i<512;i++)
       {
              tmp1 = ssp_transceive_word(tmp);
              //tmp1 = ssp_ioctl(dev_ssp, SSP_TX_RX_WORD, tmp);
              *_data++ = (mmc_dat_old<<(mmc_dat_pos)) | (tmp1>>(8-mmc_dat_pos));
              mmc_dat_old = tmp1;
       }

       //Get CRC 2 bytes
       tmp1 = ssp_transceive_word(tmp);
       tmp1 = ssp_transceive_word(tmp);
       //tmp1 = ssp_ioctl(dev_ssp, SSP_TX_RX_WORD, tmp);
       //tmp1 = ssp_ioctl(dev_ssp, SSP_TX_RX_WORD, tmp);
       //One extra for safety
       tmp1 = ssp_transceive_word(tmp);
       //tmp1 = ssp_ioctl(dev_ssp, SSP_TX_RX_WORD, tmp);

    return 512;
}

/********************************************************************
 * Function: mmc_start_sector_write
 *
```

```
 * Purpose:  Starts the write of a sector.
 *
 * Processing:  Set the sector size to '1' and issue the sector write command.
 *
 * Parameters:  None
 *
 * Outputs:  None
 *
 * Returns:  Nothing
 *
 * Notes:  None
 *
 ***********************************************************************/
void mmc_start_sector_write (void)
{
       INT_32 response;
       UNS_8 tmp, tmp1;

       // Send command to initialize the MMC card
       MMC_send_cmd(CMD_WRITE_BLOCK, mmc_data_addr);
       response = MMC_get_response(RESP_R1);

       //Wait for 8 clocks
       tmp=0xff;
       tmp1 = ssp_transceive_word(tmp);
       //tmp1 = ssp_ioctl(dev_ssp, SSP_TX_RX_WORD, tmp);
}

/***********************************************************************
 * Function: mmc_write_data
 *
 * Purpose:  Write data to the MMC card.
 *
 * Processing:  Copy a block of data of from the source address to the
 *              MMC card buffer.
 *
 * Parameters:
 *          data  : Pointer to where to get data to write to the MMC card
 *          bytes : Number of bytes to write
 *
 * Outputs:  None
 *
 * Returns:  The number of bytes written to the card.
 *
 * Notes:  This function will read out 512 bytes fixed
 *
 ***********************************************************************/
INT_32 mmc_write_data (void *data, INT_32 bytes)
```

```
{
    INT_32 i;
    INT_32 response;
    UNS_8 * _data = (UNS_8 *)data;
    UNS_8 tmp1,tmp;

       //Send data token
       tmp1 = ssp_transceive_word(DATA_TOKEN);
       //tmp1 = ssp_ioctl(dev_ssp, SSP_TX_RX_WORD, DATA_TOKEN);

       for(i=0;i<512;i++)
       {
              tmp1 = ssp_transceive_word(*_data++);
              //tmp1 = ssp_ioctl(dev_ssp, SSP_TX_RX_WORD, *_data++);
       }

       tmp = 0xff;
       //Send 2 CRC bytes
       tmp1 = ssp_transceive_word(tmp);
       tmp1 = ssp_transceive_word(tmp);
       //tmp1 = ssp_ioctl(dev_ssp, SSP_TX_RX_WORD, tmp);
       //tmp1 = ssp_ioctl(dev_ssp, SSP_TX_RX_WORD, tmp);

       response = MMC_get_response(RESP_R1);

       if(response != 0) return NO_RESPONSE;

       response = 0;

       //Wait until no more busy
       while(response == 0)
       {
              response = ssp_transceive_word(0xff);
              //response = ssp_ioctl(dev_ssp, SSP_TX_RX_WORD, 0xff);
       }

    return 512;
}

/*****************************************************************************
 * Function: mmc_erase_sector
 *
 * Purpose:  Erase MMC card sectors.
 *
 * Processing:  Erase requested number of card sectors starting with
 *              specified sector number
 *
 * Parameters:
```

```
 *          start: Starting sector number
 *          bytes : Number of sectors to erase
 *
 * Outputs:  None
 *
 * Returns:  1 for error, 0 for success
 *
 * Notes:  None
 *
 ***************************************************************************/
INT_32 mmc_erase_sector (INT_32 start_sector, INT_32 n_sectors)
{

    INT_32 response;

       MMC_send_cmd(CMD_TAG_SECTOR_START, start_sector);
       response = MMC_get_response(RESP_R1);
       if(response != 0)
       {
              return 1;//Error in MMC
       }

       MMC_send_cmd(CMD_TAG_SECTOR_END, start_sector+n_sectors);
       response = MMC_get_response(RESP_R1);
       if(response != 0)
       {
              return 1;//Error in MMC
       }

       MMC_send_cmd(CMD_ERASE, 0);
       response = MMC_get_response(RESP_R1);
       if(response != 0)
       {
              return 1;//Error in MMC
       }

       response = 0;
       //Wait until MMC is no longer busy
       while(response == 0)
       {
              response = ssp_transceive_word(0xff);
              //response = ssp_ioctl(dev_ssp, SSP_TX_RX_WORD, 0xff);
       }
    return 0;
}

/***************************************************************************
 * Function: mmc_is_card_ready
```

```
 *
 * Purpose:  Determines if the card is ready for a new command.
 *
 * Processing:  If the MMC_RDY bit in the MMC status register is set,
 *              return '1', else return a '0'.
 *
 * Parameters:  None
 *
 * Outputs:  None
 *
 * Returns:  '1' if the card is ready for a new command, otherwise '0'.
 *
 * Notes:  None
 *
 ************************************************************************/
INT_32 mmc_is_card_ready (void)
{
       INT_32 response;

       MMC_send_cmd(CMD_SEND_STATUS, 0);
       response = MMC_get_response(RESP_R2);

       if((response & R2_IN_IDLE_STATE) != 0)
       {
              return 0;
       }
       else
       {
              return 1;
       }
}

/************************************************************************
 * Function: mmc_is_card_busy
 *
 * Purpose:  Determines if the card is busy
 *
 * Processing:
 *
 * Parameters:  None
 *
 * Outputs:  None
 *
 * Returns:  '1' if the card is ready for a new command, otherwise '0'.
 *
 * Notes:  None
 *
 ************************************************************************/
```

```
INT_32 mmc_is_card_busy (void)
{
       return 0;
}

/*********************************************************************
 * Function: mmc_is_card_inserted
 *
 * Purpose:  Determines if the card is inserted
 *
 * Processing:
 *
 * Parameters:  None
 *
 * Outputs:  None
 *
 * Returns:  '1' if the card is ready for a new command, otherwise '0'.
 *
 * Notes:  None
 *
 *********************************************************************/
INT_32 mmc_is_card_inserted (void)
{
       INT_32 response,i;

       response = 0xff;

       for(i=0;i<10;i++)
       {
              MMC_send_cmd(CMD_GO_IDLE_STATE, 0);
              response = MMC_get_response(RESP_R1);
              if(response == R1_IN_IDLE_STATE)
                     return 1;
       }

       return 0;
}


/*********************************************************************
 * Function: mmc_shutdown
 *
 * Purpose:  Shutdown the MMC interface driver.
 *
 * Processing:  This function does nothing and is a placeholder.
 *
 * Parameters:  None
 *
```

```
 * Outputs:  None
 *
 * Returns:  Nothing
 *
 * Notes:  None
 *
 ***************************************************************************/
void mmc_shutdown (void)
{
    /* Do nothing */
    ;
}

/***************************************************************************
 * Function: MMC_send_cmd
 *
 * Purpose:  MCU send 6 bytes of command to MMC
 *
 * Processing:
 *
 * Parameters:  None
 *
 * Outputs:  None
 *
 * Returns:  Nothing
 *
 * Notes:  None
 *
 ***************************************************************************/
static void MMC_send_cmd(UNS_8 cmd, UNS_32 arg)
{
       CHAR cmd_dat[6], tmp, tmp1[6];
       CHAR * out, * in;

       out = cmd_dat;
       in = tmp1;

       //Construct byte 1
       tmp = cmd | _BIT(6);
       tmp &= ~(_BIT(7));
       cmd_dat[0] = tmp;

       //Construct byte 2 to 5
       cmd_dat[1] = arg>>24;
       cmd_dat[2] = arg>>16;
       cmd_dat[3] = arg>>8;
       cmd_dat[4] = arg;
```

```
       //Construct CRC
       //cmd_dat[5] = (get_crc7(cmd_dat)<<1)|1;

       if(cmd == CMD_CRC_ONOFF)
       {
              cmd_dat[5] = arg;
       }
       else
       {
              cmd_dat[5] = 0x95;
       }

       //Send the command out
    tmp = 6;
    /* Do SSP transceive under polling mode */
    while(tmp--)
    {
        /* Transceive the sentense */
        *in++ = ssp_transceive_word(*out++);
        //*in++ = ssp_ioctl(dev_ssp, SSP_TX_RX_WORD, *out++);
    }
}

/*********************************************************************
 * Function: MMC_get_response
 *
 * Purpose:  MCU receive number of bytes of response from MMC
 *
 * Processing:
 *
 * Parameters:  RESP_R1, RESP_R1B, RESP_R2, RESP_R3
 *
 * Outputs:  None
 *
 * Returns:  0xffffffff for No response
 *
 * Notes:  Return 1 byte for R1,R1B. 2 bytes for R2, 4 bytes for R3(OCR)
 *
 *********************************************************************/
static INT_32 MMC_get_response(UNS_8 type)
{
       UNS_8 i;
       volatile UNS_8 tmp, tmp1, pos;
       volatile INT_32 response;

       tmp1 = 0xff;
       response = 0;
       tmp = 0xff;
```

```
for(i=0;i<5;i++)
{
	if(tmp1 == 0xff)//Wait until receive a response
	{
		tmp1 = ssp_transceive_word(tmp);
		//tmp1 = ssp_ioctl(dev_ssp, SSP_TX_RX_WORD, tmp);
	}
	else
	{
		break;
	}
}

if(tmp1 == 0xff)
	return NO_RESPONSE;//No response

pos = 0x80;

// Find starting bit 0 position
for(i=0;i<8;i++)
{
	if((tmp1 & (pos>>i)) == 0)
	{
		pos = i;
		break;
	}
}

//tmp1 holds the first read
tmp1 = tmp1<<pos;//clear the leading 1s
response = tmp1;
response = response << (8-pos);

//Read out rest of the response
if(type == RESP_R1)
{
	tmp1 = ssp_transceive_word(tmp);
	//tmp1 = ssp_ioctl(dev_ssp, SSP_TX_RX_WORD, tmp);
	response |= tmp1;
	response = response >> (8-pos);
}
else if(type == RESP_R1B)
{
	tmp1 = ssp_transceive_word(tmp);
	//tmp1 = ssp_ioctl(dev_ssp, SSP_TX_RX_WORD, tmp);
	response |= tmp1;
	tmp1 = 0xff;
```

```
            while(tmp1 != 0xff)//Wait until no busy received
            {
                  tmp1 = ssp_transceive_word(tmp);
                  //tmp1 = ssp_ioctl(dev_ssp, SSP_TX_RX_WORD, tmp);
            }
            response = response >> (8-pos);
      }
      else if(type == RESP_R2)
      {
            tmp1 = ssp_transceive_word(tmp);
            //tmp1 = ssp_ioctl(dev_ssp, SSP_TX_RX_WORD, tmp);
            response |= tmp1;
            response = response<<8;
            tmp1 = ssp_transceive_word(tmp);
            //tmp1 = ssp_ioctl(dev_ssp, SSP_TX_RX_WORD, tmp);
            response |= tmp1;
            response = response >> (8-pos);
      }
      else if(type == RESP_R3)
      {
            tmp1 = ssp_transceive_word(tmp);
            //tmp1 = ssp_ioctl(dev_ssp, SSP_TX_RX_WORD, tmp); //OCR3
            response |= tmp1;
            response = response<<8;
            tmp1 = ssp_transceive_word(tmp);
            //tmp1 = ssp_ioctl(dev_ssp, SSP_TX_RX_WORD, tmp); //OCR2
            response |= tmp1;
            response = response<<8;
            tmp1 = ssp_transceive_word(tmp);
            //tmp1 = ssp_ioctl(dev_ssp, SSP_TX_RX_WORD, tmp); //OCR1
            response |= tmp1;
            response = response<<8;
            tmp1 = ssp_transceive_word(tmp);
            //tmp1 = ssp_ioctl(dev_ssp, SSP_TX_RX_WORD, tmp); //OCR0
            response |= tmp1;
            response = response<<8;
            tmp1 = ssp_transceive_word(tmp);
            //tmp1 = ssp_ioctl(dev_ssp, SSP_TX_RX_WORD, tmp); //OCR0
            response |= tmp1;
            response = response >> (8-pos);
      }
      else
      {
            response = NO_RESPONSE;
      }

      return response;
}
```

```
/*********************************************************************
 * Function: spi_delay_bytes
 *
 * Purpose:  SPI delay number of 8 clocks
 *
 * Processing:
 *
 * Parameters:  None
 *
 * Outputs:  None
 *
 * Returns:  Nothing
 *
 * Notes:  None
 *
 *********************************************************************/
static void spi_delay_bytes(INT_32 n_8_clks)
{
       CHAR tmp1,tmp2;

       tmp2 = 0xff;
    while(n_8_clks--)
    {
        /* Transceive the sentense */
             tmp1 = ssp_transceive_word(tmp2);
        //tmp1 = ssp_ioctl(dev_ssp, SSP_TX_RX_WORD, tmp2);
    }
}

/*********************************************************************
 * Function: get_crc7
 *
 * Purpose:  Get CRC 7 value
 *
 * Processing:
 *
 * Parameters:  None
 *
 * Outputs:  None
 *
 * Returns:  Nothing
 *
 * Notes:  Taken from Siemens MMC application note
 *
 *********************************************************************/
#define CMD_BYTE_LENGTH6
static CHAR get_crc7(CHAR cmd_bytes[])
```

```
{
       CHAR byte, ibit;
       CHAR reg = 0;

       //for (byte = CMD_BYTE_LENGTH-1; byte > 0; byte--)
       for (byte = 0; byte < CMD_BYTE_LENGTH-1; byte++)
       {
              for (ibit=0; ibit<8; ibit++)
              {
                     reg <<= 1;
                     reg ^= ((((cmd_bytes[byte] << ibit) ^ reg) & 0x80) ? 0x9 : 0);
              }
       }
       return reg;
}

/*********************************************************************
 * Function: get_crc16
 *
 * Purpose:  Get CRC 16 value
 *
 * Processing:
 *
 * Parameters:  None
 *
 * Outputs:  None
 *
 * Returns:  Nothing
 *
 * Notes:  Taken from Siemens MMC application note
 *
 *********************************************************************/
#define D_CRC_LEN6
#define D_CRC_POLYN0x11021
#define D_CRC_HIGHBIT0x10000

static UNS_16 get_crc16(CHAR * uc_data, INT_32 data_length)
{
       INT_32 byte;
       CHAR   c_bit;
       UNS_16 reg = 0;

       for (byte = 0; byte < data_length; byte++)
       {
              for (c_bit=0; c_bit<8; c_bit++)
              {
                     reg <<= 1;
                     reg ^= (( (((UNS_32)uc_data[byte] << (c_bit + (D_CRC_LEN -
```

```
                        7))) ^ reg) & D_CRC_HIGHBIT) ? D_CRC_POLYN : 0);
            }
      }
      return reg;
}


static INT_32 ssp_transceive_word(INT_32 data)
{
      INT_32 status;
    /* wait untile transmit fifo is not full */
    while((SSP->sr & SSP_SR_TNF) == 0);
    SSP->dr = (UNS_16)data;
    /* wait until receive fifo is not empty */
    while((SSP->sr & SSP_SR_RNE) == 0);
    status = SSP->dr;
    return status;
}


#ifndef _LH79520_LPD_MMC_DRIVER_H
#define _LH79520_LPD_MMC_DRIVER_H

#ifdef __cplusplus
#if __cplusplus
extern "C"
{
#endif // __cplusplus
#endif // __cplusplus

#include "abl_types.h"

// Command list for MMC operation
#define CMD_GO_IDLE_STATE0 //Reset the MMC
#define      CMD_SEND_OP_COND1   //Activate initialization process
#define CMD_SEND_STATUS    13     //Request card send status
#define      CMD_SET_BLOCKLEN16  //Set block length (in bytes)
#define      CMD_READ_BLOCK      17     //Read a block of data
#define      CMD_WRITE_BLOCK     24     //Write a block of data
#define      CMD_TAG_SECTOR_START32//Set first sector for erase
#define      CMD_TAG_SECTOR_END33//Set last sector for erase
#define      CMD_UNTAG_SECTOR34  //Remove one selected sector
                                        //from erase group
#define      CMD_ERASE                  38//Erase all selected sectors
#define      CMD_CRC_ONOFF       59     //0 to turn off CRC, 1 to turn on CRC

// Response type
#define RESP_R1                         1//R1 type response 1byte
```

```
#define      RESP_R2                    2//R2 type response 2bytes
#define RESP_R1B                3      //R1B type response 1byte + busy bytes
#define      RESP_R3                    4//R3 type response 5 bytes
#define      NO_RESPONSE                0xffffffff

// R1 type response bits defintion
#define      R1_IN_IDLE_STATE_BIT(0)
#define      R1_ERASE_RESET      _BIT(1)
#define      R1_ILLEGAL_CMD      _BIT(2)
#define R1_CRC_ERROR       _BIT(3)
#define      R1_ERASE_SEQ_ERROR_BIT(4)
#define      R1_ADDR_ERROR       _BIT(5)
#define      R1_PARAMETER_ERROR_BIT(6)

//R2 type response bits definition
#define      R2_CARD_LOCKED      _BIT(0)
#define      R2_WP_ERASE_SKIP_BIT(1)
#define      R2_ERROR                   _BIT(2)
#define      R2_CC_ERROR                _BIT(3)
#define      R2_CARD_ECC_FAILED_BIT(4)
#define      R2_WP_VIOLATION     _BIT(5)
#define      R2_ERASE_PARAM      _BIT(6)
#define      R2_OUT_OF_RANGE     _BIT(7)
#define      R2_IN_IDLE_STATE_BIT(8)
#define      R2_ERASE_RESET      _BIT(9)
#define      R2_ILLEGAL_CMD      _BIT(10)
#define R2_CRC_ERROR       _BIT(11)
#define      R2_ERASE_SEQ_ERROR_BIT(12)
#define      R2_ADDR_ERROR       _BIT(13)
#define      R2_PARAMETER_ERROR_BIT(14)

//Data response bits definition
#define      DATA_RESP_MASK      0x1F   //Data block written will
                                        //be acknowledged by token
#define      DATA_RESP_ACCEPTED0X15
#define      DATA_RESP_REJECTED0X1B

//Data Token
#define      DATA_TOKEN                 0xFE//First byte send when
                                        //data read and write

//Data Error Token
#define      DATA_ERROR                 _BIT(0)//Error bits when read fails
#define      DATA_CC_ERROR       _BIT(1)
#define      DATA_CARD_ECC_FAILED_BIT(2)
#define      DATA_OUT_OF_RANGE_BIT(3)

//*********************************************************************
```

```
// Card functions
//*********************************************************************

INT_32 mmc_init (void);
void   mmc_shutdown (void);
INT_32 mmc_is_card_ready (void);
INT_32 mmc_is_card_busy (void);
INT_32 mmc_is_card_inserted (void);
void   mmc_set_sector (UNS_32 sectorno);
void mmc_start_sector_read (void);
void mmc_start_sector_write (void);
INT_32 mmc_read_data (void *data, INT_32 bytes);
INT_32 mmc_write_data (void *data, INT_32 bytes);
INT_32 mmc_erase_sector (INT_32 start_sector, INT_32 n_sectors);

#ifdef __cplusplus
}
#endif
#endif // LH79520_MMC_DRIVER_H
```

REFERENCE

The MultiMediaCard System Specification, Version 3.2 - by the MMCA Technical Committee

The LH79520 Universal Microcontroller User's Guide - by NXP Semiconductors

ANNEX A: Disclaimers (11)

1. t001dis100.fm: General (DS, AN, UM, errata)

General — Information in this document is believed to be accurate and reliable. However, NXP Semiconductors does not give any representations or warranties, expressed or implied, as to the accuracy or completeness of such information and shall have no liability for the consequences of use of such information.

2. t001dis101.fm: Right to make changes (DS, AN, UM, errata)

Right to make changes — NXP Semiconductors reserves the right to make changes to information published in this document, including without limitation specifications and product descriptions, at any time and without notice. This document supersedes and replaces all information supplied prior to the publication hereof.

3. t001dis102.fm: Suitability for use (DS, AN, UM, errata)

Suitability for use — NXP Semiconductors products are not designed, authorized or warranted to be suitable for use in medical, military, aircraft, space or life support equipment, nor in applications where failure or malfunction of a NXP Semiconductors product can reasonably be expected to result in personal injury, death or severe property or environmental damage. NXP Semiconductors accepts no liability for inclusion and/or use of NXP Semiconductors products in such equipment or applications and therefore such inclusion and/or use is at the customer's own risk.

4. t001dis103.fm: Applications (DS, AN, UM, errata)

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

5. t001dis104.fm: Limiting values (DS)

Limiting values — Stress above one or more limiting values (as defined in the Absolute Maximum Ratings System of IEC 60134) may cause permanent damage to the device. Limiting values are stress ratings only and operation of the device at these or any other conditions above those given in the Characteristics sections of this document is not implied. Exposure to limiting values for extended periods may affect device reliability.

6. t001dis105.fm: Terms and conditions of sale (DS)

Terms and conditions of sale — NXP Semiconductors products are sold subject to the general terms and conditions of commercial sale, as published at http://www.nxp.com/profile/terms, including those pertaining to warranty, intellectual property rights infringement and limitation of liability, unless explicitly otherwise agreed to in writing by NXP Semiconductors. In case of any inconsistency or conflict between information in this document and such terms and conditions, the latter will prevail.

7. t001dis106.fm: No offer to sell or license (DS)

No offer to sell or license — Nothing in this document may be interpreted or construed as an offer to sell products that is open for acceptance or the grant, conveyance or implication of any license under any copyrights, patents or other industrial or intellectual property rights.

8. t001dis107.fm: Hazardous voltage (DS, AN, UM, errata; if applicable)

Hazardous voltage — Although basic supply voltages of the product may be much lower, circuit voltages up to 60 V may appear when operating this product, depending on settings and application. Customers incorporating or otherwise using these products in applications where such high voltages may appear during operation, assembly, test etc. of such application, do so at their own risk. Customers agree to fully indemnify NXP Semiconductors for any damages resulting from or in connection with such high voltages. Furthermore, customers are drawn to safety standards (IEC 950, EN 60 950, CENELEC, ISO, etc.) and other (legal) requirements applying to such high voltages.

9. t001dis108.2.fm: Bare die (DS; if applicable)

Bare die (if applicable) — Products indicated as Bare Die are subject to separate specifications and are not tested in accordance with standard testing procedures. Product warranties and guarantees as stated in this document are not applicable to Bare Die Products unless such warranties and guarantees are explicitly stated in a valid separate agreement entered into by NXP Semiconductors and customer.

10. t001dis109.fm: AEC unqualified products (DS, AN, UM, errata; if applicable)

AEC unqualified products — This product has not been qualified to the appropriate Automotive Electronics Council (AEC) standard Q100 or Q101 and should not be used in automotive critical applications, including but not limited to applications where failure or malfunction of an NXP Semiconductors product can reasonably be expected to result in personal injury, death or severe property or environmental damage. NXP Semiconductors accepts no liability for inclusion and/or use of NXP Semiconductors products in such equipment or applications and therefore such inclusion and/or use is for the customer’s own risk.

11. t001dis110.fm: Suitability for use in automotive applications only (DS, AN, UM, errata; if applicable)

Suitability for use in automotive applications only — This NXP Semiconductors product has been developed for use in automotive applications only. The product is not designed, authorized or warranted to be suitable for any other use, including medical, military, aircraft, space or life support equipment, nor in applications where failure or malfunction of an NXP Semiconductors product can reasonably be expected to result in personal injury, death or severe property or environmental damage. NXP Semiconductors accepts no liability for inclusion and/or use of NXP Semiconductors products in such equipment or applications and therefore such inclusion and/or use is at the customer’s own risk.