Programming the TMS320VC5509 Multi Media Controller in Native Mode

Rishi Bhattacharya — *C5000 DSP Software Applications*

ABSTRACT

This document contains information and examples on accessing MMC/SD FLASH 7 cards through the C5509 MMC Controller operating in the native mode. The document provides an overview of the native mode functionality, including programming flow for the typical tasks that the user encounters when interfacing to the MMC cards in the native mode. For each operation, the procedures are highlighted with tables of relevant registers, register fields, and code listings. Project collateral described in this application report can be downloaded from http://www.ti.com/lit/zip/SPRA808.

Contents

List of Figures

List of Tables

1 MMC Native Mode Overview

The MMC controller, internal to the DSP, is accessed through the peripheral bus with either the CPU or DMA. All MMC operations are implemented through a set of registers inside the controller. There are two different modes in which the MMC controller operates:

- native mode
- SPI mode

The mode of operation is set immediately following a reset. Both modes support two types of cards:

- MMC
- SD

Figure 1 shows the DSP based MMC controller, the MMC/SD card external to the DSP, and the MMC bus connecting the controller with the MMC/SD card.

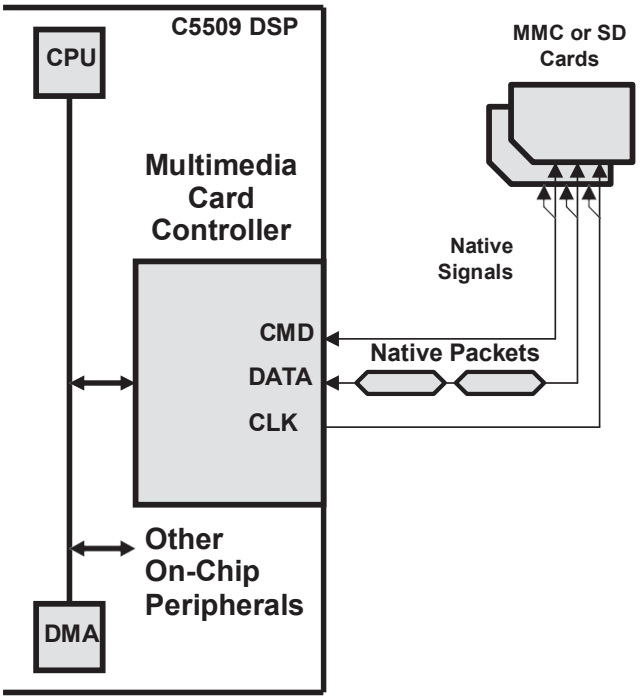

Figure 1. C5509 MMC Interface in Native Mode

1.1 MMC Controller Registers

There are six groups of registers inside the MMC controller.

- **Control registers** control the operation of the MMC controller including clock settings.
- **Status registers** reflect the status of the controller, card, and the serial Interface.
- **Block registers** are used to set up block transfers.
- **Command registers** trigger the controller and the card to execute data transfer commands.
- **Response registers** hold the card's responses to the commands.
- **Data registers** hold the incoming and outgoing data.

Figure 2 shows the MMC controller register organized into six distinct groups.

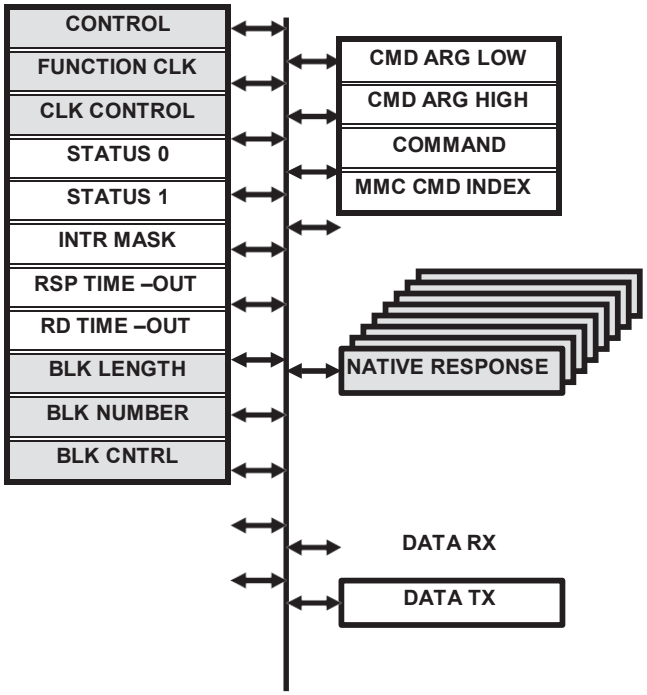

Figure 2. MMC Controller Registers

1.1.1 Control Registers

The CONTROL register configures the operation of the MMC controller, including native or SPI mode selection, MMC or SD mode selection, and peripheral reset.

The FUNCTION CLOCK control register sets the controller logic operating frequencyby dividing down the CPU clock, .

The CLOCK control register sets the frequency of the serial interface's CLK pin. This determines the maximum data rate between the card and the controller. This clock rate is generated by dividing down the controller clock.

1.1.2 Status Registers

The STATUS0 and STATUS1 registers reflect the status of the controller's operations. This includes Data Rx/Tx Ready, command response, data response, CRC error, time-out error, etc. Transition of the status register bits can trigger the peripheral interrupt to CPU/DMA if mask enabled.

The INTERRUPT MASK register determines which events in the two status registers are allowed to trigger a Peripheral Interrupt to the CPU/DMA. There is only one interrupt driven by this peripheral and it is the CPU's function to poll the Status Registers to find out what caused the interrupt (if more than 1 is enabled).

The COMMAND RESPONSE TIME-OUT register triggers an event in the status register if a card is not responding to a command.

The DATA READ TIME-OUT register triggers an event in the status register if the card is not sending data soon enough after receiving the read command.

1.1.3 Block Registers

The BLOCK LENGTH register reflects the status of controller's operations such as Data Rx/Tx Ready, command response, data response, CRC error, time-out error, etc. The transition of status register bits can trigger the peripheral interrupt to CPU/DMA if mask enabled. Typical block length is 512 bytes.

The NUMBER OF BLOCKS register specifies the total number of blocks to be transferred between the card and the controller by multi-block commands such as READ_MULTIPLE_BLOCKS, or WRITE_MULTIPLE_BLOCKS. Multi-block commands can only be used in the native mode and are not allowed in SPI mode.

The NUMBER OF BLOCKS COUNTER register counts the number of blocks transferred during multi-block commands. The CPU reads the counter and determines the number of blocks left to be transferred.

1.1.4 Command Registers

The COMMAND ARGUMENT LOW register represents the low 8 bits of command argument. Command argument registers are loaded before the command register.

The COMMAND ARGUMENT HIGH register represents the high 8 bits of command argument. Command argument registers are loaded before the command register.

The COMMAND register triggers the controller to start data transfers between the controller and the card. The command argument registers must be loaded prior to writing to the command register.

The COMMAND INDEX register stores the first byte of a command response from the card, following start of command by the controller. With this option, the card notifies the DSP that it has just executed the command. This register occurs only in Native mode.

1.1.5 Response Registers

The NATIVE RESPONSE registers 0-7 hold the command response from the card, following the issue of command to the card by the controller. The lengths of responses vary between commands and modes.

1.1.6 RX/TX DATA Registers

The DATA RECEIVE register temporarily stores the data received from a card during block reads. From this register, a card is read by the CPU or DMA, and copied into the DSP memory. The CPU polls the Data Rx ready bit in the status register to detect new data in the data receive register. The DMA relies on the interrupt triggered by the same bit to perform a transfer.

The DATA TRANSMIT register holds data from the DSP for the DMA or CPU during block writes to the card. Each word written into the data transmit register is transferred out by the controller to the card. The CPU polls the Data Tx Ready bit in the status register to detect data that has been transferred out of the card. DMA relies on the interrupt triggered by the same bit to perform a transfer.

1.2 Serial Interface Bus Between the Controller and the Card

All communication between the controller and the card takes place in the form of tokens. Before communication can take place, the controller broadcasts a command to all cards instructing them to identify themselves. Commands from the controller to the card are followed by arguments and are acknowledged by the card with command response tokens. In contrast to the SPI mode, an active card is selected with a command instead of a hard-wired chip select. All commands and command response tokens are transferred on the bi-directional CMD signal. All data and data response tokens are transmitted over the bi-directional data signal(s).

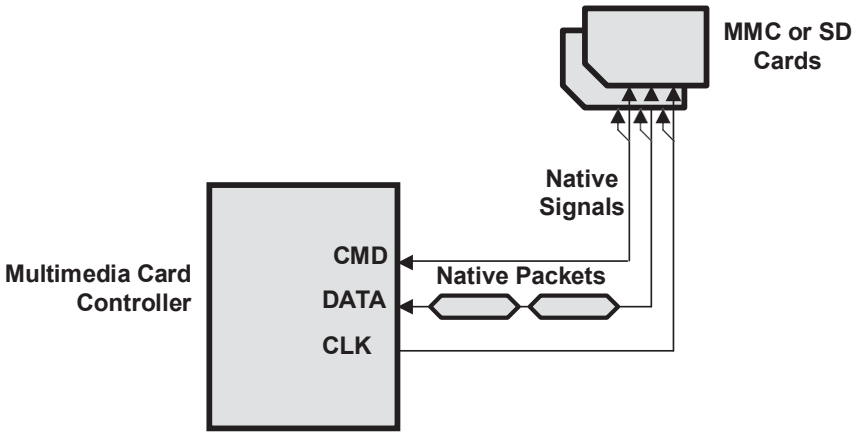

Figure 3. Native Mode Serial Interface

Figure 3 represents one bi-directional data signal, clock signal and one bi-directional command signal per card.

In the native mode, the MMC controller communicates with one or more MMC cards across a 3-pin serial interface. SD interface features three additional data pins for a total of four data pins. The clock pin drives the transfer clock that determines the rate at which the data is being transferred to and from the card across the DATA signal(s). The CMD signal carries commands from the controller to the card and the command responses from the card to the controller.

1.3 Internal Card Structure

The MMC and SD cards contain FLASH storage media, FLASH interface, a microcontroller for programming the FLASH, and a set of registers. The DSP typically does not have direct access to any of the MMC/SD card components. All communication between the card and the controller is implemented by the DSP accessing the controller registers and not the card registers.

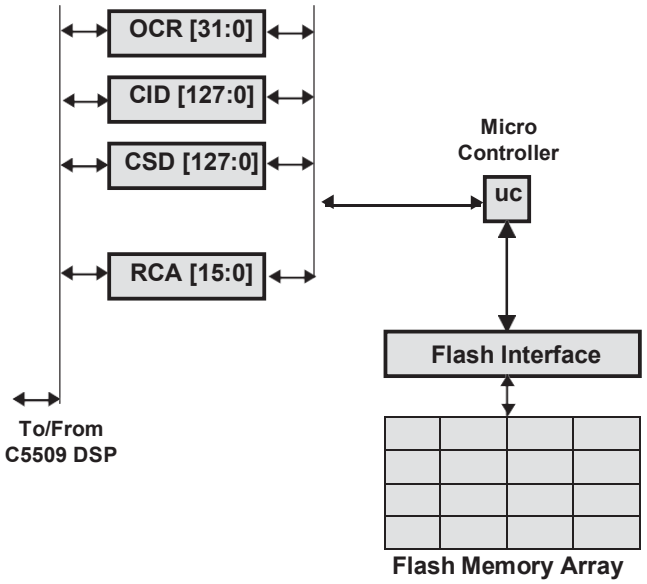

Figure 4. Internal MMC/SD Card Structure

The OCR, CID and CSD registers carry the card configuration information.

OCR – Operating conditions register, can be read by the SEND_OP_COND command.

CID – Manufacturing data including card ID, serial number, date, revision, etc. Card ID is read by the SEND_CID command.

CSD – Card-specific data contains all the configuration information required to access the card data, like protocol version, access time, max data rates, block lengths, max current consumption, etc. Card-specific data can be read by the send CSD command.

The RCA register holds the card-relative communication address for the current session. This register is programmed with SET_RELATIVE_ADDR command from the DSP to the controller.

2 Initialized Operation

To write a single block of data in native mode, the CPU must first identify which card is to be programmed with the data, as one controller can support multiple cards. To select a card, the CPU loads the controller argument registers with the relative card address (RCA), and the command register with the SEL_DESEL_CARD command. The execution of the SEL_DESEL_CARD command by the controller activates the card with the matching RCA and deselects all other cards on the serial interface.

Next, the CPU loads the starting address of the block to be programmed into FLASH, and to low and high argument registers inside the controller. The first byte of the 512 byte block of data is loaded to the Data Tx register. Then the WRITE_BLOCK command is loaded to the command register to trigger the controller to start the block transfer from the DSP to the card.

During the block transfer, the CPU periodically examines the controller STATUS0 register for DATA TX empty status before loading the DATA TX register with the next byte of the block. The CPU also examines the STATUS0 register to detect the end of transfer and to identify any CRC errors that may have occurred during transfer of data from the controller to the card.

2.1 MMC Controller Initialization

Figure 5 shows a block diagram of the MMC_initNative function. This function is used to configure the MMC controller for data transfer in the native mode.

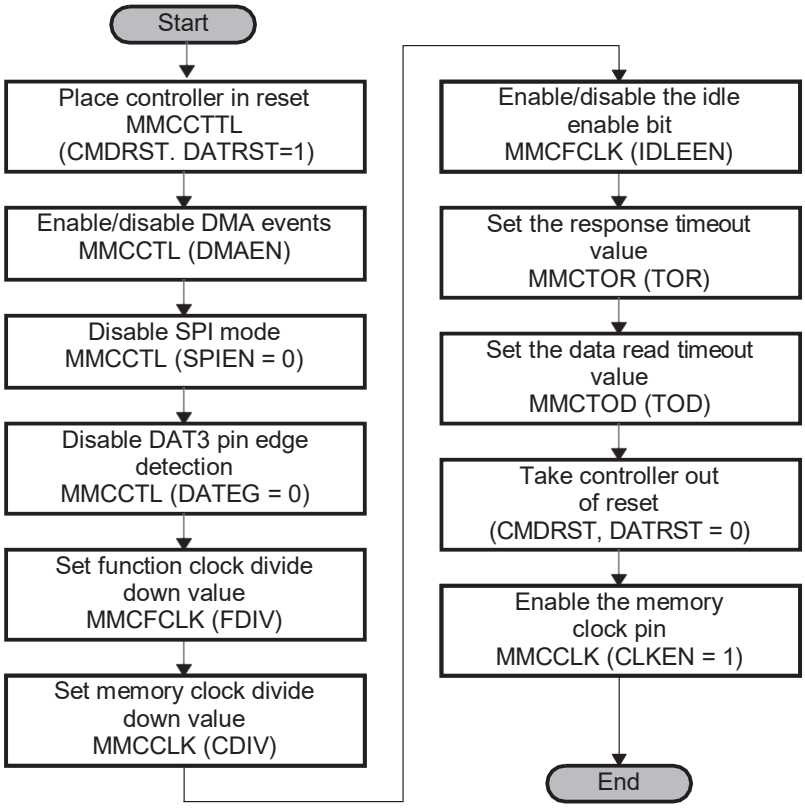

Figure 5. MMC Controller Configuration Configuration (MMC_initNative)

This function takes an MMC handle and a structure that contains various parameters as input and configures the MMC controller. The various parameters within the structure include DMA event enable/disable, idle enable/disable, CPU clock to function clock divide down value, function clock to memory clock divide down value, response timeout value, data read timeout value, and the block length.

Initially, the function places the controller in reset by setting the CMDRST and DATRST bits in the MMCCTL register. It then enables or disables DMA events by appropriately setting the DMAEN bit in the MMCCTL register. Then, the function disables SPI mode and edge detection on the DAT3 pin by clearing the SPIEN and DATEG bits in the MMCCTL register, respectively.

Next, the function configures the CPU clock to function clock divide down value by placing the value passed in the FDIV field of the MMCFCLK register. It then proceeds to do the same for the function clock to memory clock divide down value by placing the value passed in the CDIV field of the MMCCLK register.

The function continues to configure the MMC controller by enabling or disabling the idling capabilities of the MMC controller. This is done by setting the IDLEEN bit in the MMCFLCK register appropriately. Next, the response timeout and data read timeout values are placed in the MMCTOR and MMCTOD registers, respectively.

Finally, the function takes the controller out of reset by clearing the CMDRST and DATRST bits in the MMCCTL register and enables the memory clock by setting the CLKEN bit in the MMCCLK register.

2.2 Card Initialization

Figure 6 illustrates a block diagram for the MMC card initialization procedure in native mode. It is imperative that the procedure be followed in the order demonstrated in the figure.

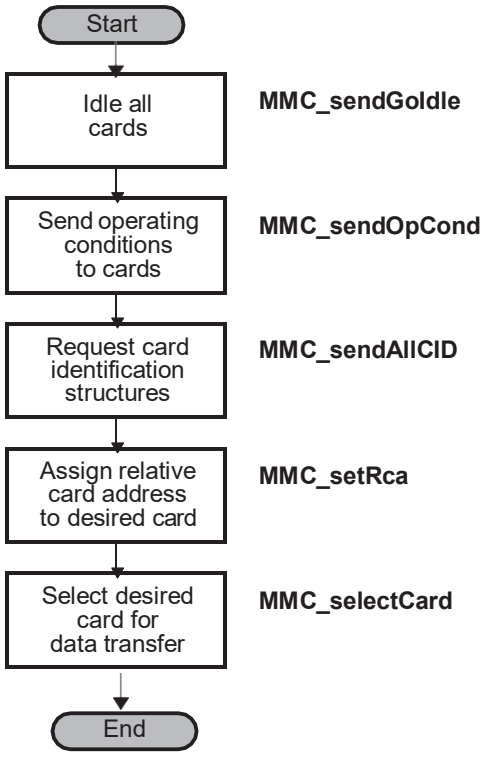

Figure 6. MMC Card Initialization Procedure (Native Mode)

The first step in initializing a card for data transfer in native mode is to idle all cards, or in other words, to perform a software reset. This can be accomplished by calling the MMC_sendGoIdle function. The function issues the GO_IDLE_STATE command to all cards and takes only the MMC handle as an argument.

Example:

```
MMC_sendGoIdle(mmc0);  /* idle all cards */
```

Next, it is necessary to inform all cards of the operating voltage conditions. This can be achieved by calling the MMC_sendOpCond function. This function takes an MMC handle and a mask of acceptable voltage ranges as arguments. Any cards which cannot operate in the desired voltage range will be instructed to go into an inactive state.

Example:

```
temp=MMC_sendOpCond(mmc0,0x00100000);   /* 3.2 – 3.3 [V] window */
```

The next step in the card initialization process is to instruct all cards to send their card identification structures. This is accomplished by calling the MMC_sendAllCID function. This function takes an MMC handle and a pointer to an MMC_CardIDObj structure as arguments.

Example:

```
temp = MMC_sendAllCID(mmc0,&cardid); /* instruct cards to send CID
numbers */
```

After obtaining each card's unique card identification number, it is the controller's responsibility to assign the card a much shorter relative card address (RCA). This can be achieved by calling the MMC_setRca function. This function takes an MMC handle, a pointer to a MMC_CardObj structure, and the value of the RCA as arguments.

Example:

```
temp = MMC_setRca(mmc0,&card,1);    /* assign relative card address of 1 */
```

Finally, after assigning each card a unique RCA, it is necessary to select the desired card by issuing the SELECT_CARD command. This is achieved by calling the MMC_selectCard function. This function takes an MMC handle and a pointer to an MMC_CardObj structure as arguments. The MMC_CardObj structure must be the same as the one that was used to call the MMC_setRca function in the previous step.

Example:

```
temp = MMC_selectCard(mmc0,&card);  /* select desired card for data
transfer */
```

3 Single Block Write/Read Operation

This application note has shown how to perform typical operations involving the MMC peripheral operating in the Native mode. Each example contains a summary of how the operation is implemented inside the MMC controller, followed by a list of relevant registers, fields and C code source using MMC controller API routines from the CSL.

3.1 Single Block Write/Read Example

This example demonstrates how to write a single block of data to an MMC card in native mode. It can be used on the Spectrum Digital C5509 EVM board.

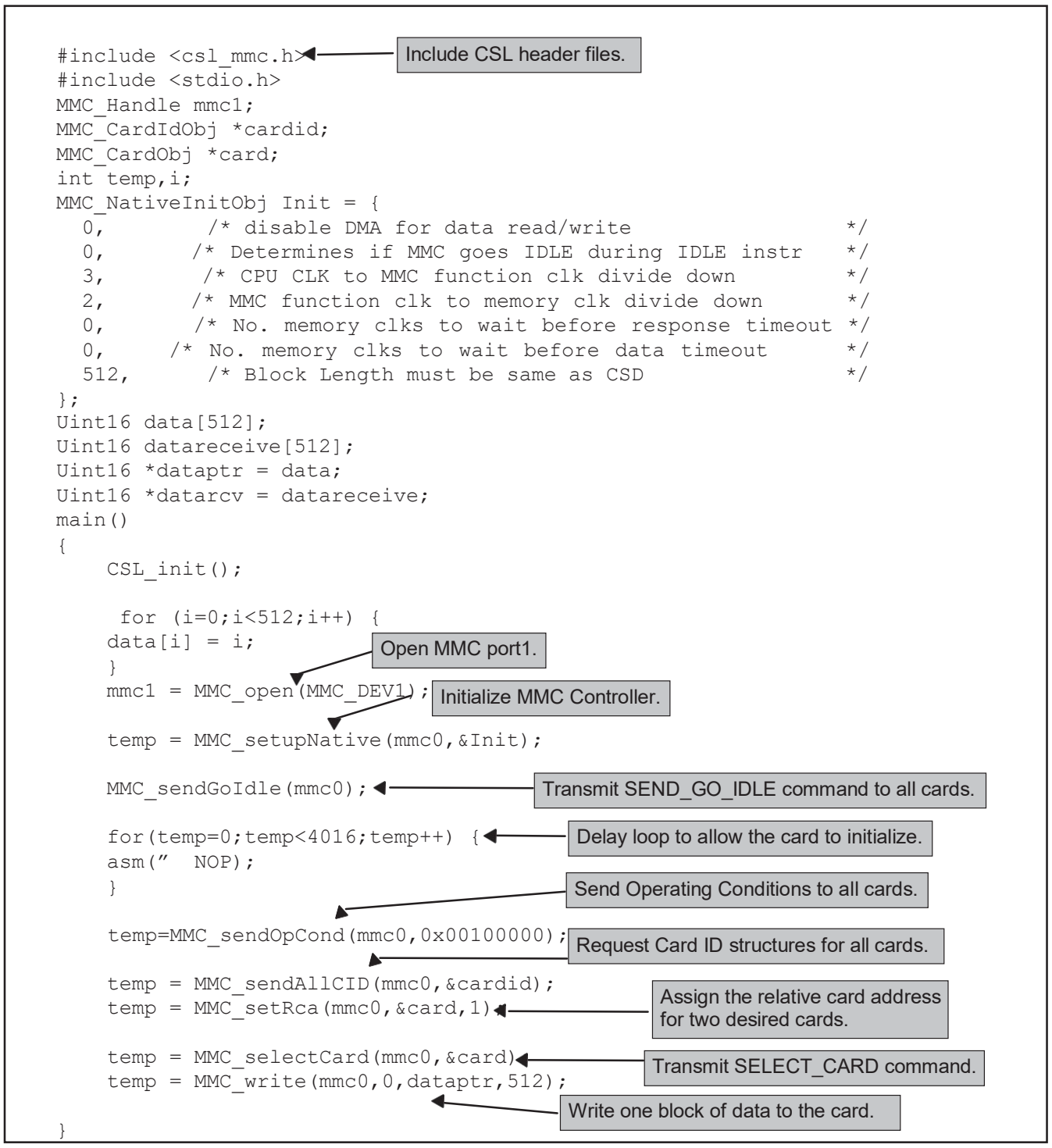

Figure 7. Sample Code Used to Write a Single Block of Data in Native Mode

3.2 Code Description

- **Step 1**: Configuring the MMC Controller for data transfer

Begin by initializing various parameters that will be used to perform the configuration. First, initialize an MMC handle.

```
MMC_Handle mmc0;
```

Next, initialize the transmit data array.

```
Uint16 data[512];
```

Call the MMC_open function to open the desired MMC port (0 or 1). This function returns a handle which is used in all future communications with this port. For this example, port 1 has been selected arbitrarily.

```
mmc1 = MMC_open(MMC_DEV0);
```

Finally, call MMC_init to configure the MMC controller for data transfer in native mode.

```
temp = MMC_init(mmc0,&Init);
```

The following table describes the parameters that are included in the MMC_NativeInitObj structure:

Table 1. MMC_NativeInitObj Structure Parameters

0	Disable DMA events
0	CPU Determines if MMC goes IDLE during IDLE instr
2	MCLK to MMC function clk divide down
3	MC function clk to memory clk divide
0	No. memory clks to wait before response timeout
0	No. memory clks to wait before data timeout
512	Sets Block Length (bytes)

- **Step 2**: Configuring the MMC Card for data transfer

In this step, initialize the MMC card for data transfer. The card must be inserted into the MMC slot before proceeding with the following steps. Call MMC_sendGoIdle with the handle as an argument in order to set all cards in the idle state.

```
MMC_sendGoIdle(mmc0);
```

Next, call the MMC_sendOpCond function to send the operating voltage conditions (3.2 – 3.3 V) to the card. For more information on the arguments to this function, please refer to the

```
temp=MMC_sendOpCond(mmc0,0x00100000);
```

The next step in the card initialization process is to instruct all cards to send their Card Identification structures. This is accomplished by calling the MMC_sendAllCID function.

```
temp = MMC_sendAllCID(mmc0,&cardid);
```

After obtaining each card's unique card identification number, it is the controller's responsibility to assign the card a much shorter relative card address (RCA). Therefore we call the MMC_setRca function to assign the relative address of 1 to the desired card.

```
temp = MMC_setRca(mmc0,&card,1);
```

In the last step of the card initialization process, the desired card must be selected by issuing the SELECT_CARD command. This is achieved by calling the MMC_selectCard function.

```
temp = MMC_selectCard(mmc0,&card);
```

- **Step 3**: Performing the write operation

With the difficult tasks of this example completed, a simple call to MMC_write allows you to write to the MMC card. Call MMC_write with the handle (mmc1), the address on the card you wish to write to (0), a pointer to the transmit data array (dataptr), and the length of the data (512 bytes).

```
temp = MMC_write(mmc0,0,dataptr,512);
```

4 Conclusion

This application note has demonstrated typical operations involving the MMC peripheral operating in the native mode. Each example contains a summary of how the operation is implemented inside the MMC Controller, followed by a list of relevant registers, fields, and C code source using MMC controller API routines from the CSL.

IMPORTANT NOTICE AND DISCLAIMER

TI PROVIDES TECHNICAL AND RELIABILITY DATA (INCLUDING DATASHEETS), DESIGN RESOURCES (INCLUDING REFERENCE DESIGNS), APPLICATION OR OTHER DESIGN ADVICE, WEB TOOLS, SAFETY INFORMATION, AND OTHER RESOURCES “AS IS” AND WITH ALL FAULTS, AND DISCLAIMS ALL WARRANTIES, EXPRESS AND IMPLIED, INCLUDING WITHOUT LIMITATION ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT OF THIRD PARTY INTELLECTUAL PROPERTY RIGHTS.

These resources are intended for skilled developers designing with TI products. You are solely responsible for (1) selecting the appropriate TI products for your application, (2) designing, validating and testing your application, and (3) ensuring your application meets applicable standards, and any other safety, security, or other requirements. These resources are subject to change without notice. TI grants you permission to use these resources only for development of an application that uses the TI products described in the resource. Other reproduction and display of these resources is prohibited. No license is granted to any other TI intellectual property right or to any third party intellectual property right. TI disclaims responsibility for, and you will fully indemnify TI and its representatives against, any claims, damages, costs, losses, and liabilities arising out of your use of these resources.

TI’s products are provided subject to TI’s Terms of Sale (www.ti.com/legal/termsofsale.html) or other applicable terms available either on ti.com or provided in conjunction with such TI products. TI’s provision of these resources does not expand or otherwise alter TI’s applicable warranties or warranty disclaimers for TI products.

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265