F. Foust
Dept. of Electrical and Computer Engineering

Abstract

Secure Digital (SD) cards are removable flash-based storage devices that are gaining in popularity in small consumer devices such as digital cameras, PDAs, and portable music devices. Their small size, relative simplicity, low power consumption, and low cost make them an ideal solution for many applications.

This application note describes the implementation of an SD Card interface for the Texas Instruments MSP430, a low-power 16-bit microcontroller [4]. This interface, combined with the MSP430, can form the foundation for a low-cost, long-life data logger or media player or recorder.

SD Card Standard

The SD card standard is a standard for removable memory storage designed and licensed by the SD Card Association [3]. The SD Card standard is largely a collaborative effort by three manufacturers, Toshiba, SanDisk, and MEI [2] and grew out of an older standard, MultiMediaCard (MMC). The card form factor, electrical interface, and protocol are all part of the SD Card specification. The SD standard is not limited to removable memory storage devices and has been adapted to many different classes of devices, including 802.11 cards, bluetooth devices, and modems [3].

Comparison with Other Technologies

SD is one of many different types of removable memory storage devices. Among the other competing standards are CF, CF+, Sony Memory Stick, and USB. These devices all perform similar functions, but differ widely in form factors, complexity, and power consumption.

SD Cards measure only 32x24 mm. This is very small compared to most competing technologies, but is both an advantage and a disadvantage, since the small size and weight requirements cannot accommodate microdrives. However, if size is a significant design consideration, SD is an ideal choice.

The SD Card electrical interface is relatively simple, requiring at most only 6 wires for communications, while still supporting data rates in the Mbps range. Compared to USB and CF/CF+, the SD physical interface is very simple, a strong consideration if interface complexity is a concern.

SD Cards typically draw no more than 100 mA of current while active, generally less than that drawn by CF or USB devices. If power consumption is important, SD again is a good choice.

SD Electrical Interface

A diagram of an SD card is shown in Figure 1.

F. Foust
Dept. of Electrical and Computer Engineering

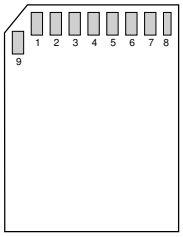

Figure 1: SD Card Diagram [2].

Table 1 lists the pin assignments for the SD Card.

Pin	Name	Function (SD Mode)	Function (SPI Mode)
1	DAT3/CS	Data Line 3	Chip Select/Slave Select ($\overline{SS}$)
2	CMD/DI	Command Line	Master Out Slave In (MOSI)
3	VSS1	Ground	Ground
4	VDD	Supply Voltage	Supply Voltage
5	CLK	Clock	Clock (SCK)
6	VSS2	Ground	Ground
7	DAT0/DO	Data Line 0	Master In Slave Out (MISO)
8	DAT1/IRQ	Data Line 1	Unused or IRQ
9	DAT2/NC	Data Line 2	Unused

Table 1: SD Card Pin Assignments [2].

From Table 1, it is apparent that many SD Card pins are dual-purpose. SD Cards support three protocols, two classes of which are distinct from each other. The descriptions listed describe the pin functions for each of these two major modes, SD mode and SPI mode.

The SD 1-bit protocol is a synchronous serial protocol with one data line, used for bulk data transfers, one clock line for synchronization, and one command line, used for sending command frames. The SD 1-bit protocol explicitly supports bus sharing. A simple single-master arbitration scheme allows multiple SD cards to share a single clock and DAT0 line.

The SD 4-bit protocol is nearly identical to the SD 1-bit protocol. The main difference is the bus width – bulk data transfers occur over a 4-bit parallel bus instead of a single wire. With proper design, this has the potential to quadruple the throughput for bulk data transfers. Both the SD 1-bit and 4-bit protocols by default require CRC protection of bulk data transfers. A CRC, or Cyclic Redundancy Check, is a simple method for detecting the presence of simple bit-inversion errors in a transmitted block of data.

In SD 4-bit mode, the input data is multiplexed over the four bus (DAT) lines and the 16-bit CRC is calculated *independently* for each of the four lines. In an all-software implementation, calculating the CRC under these conditions can be so complex that the computational overhead may mitigate the benefits of the wider 4-bit bus. A 4-bit parallel CRC is trivial to implement in hardware, however, so custom ASIC or programmable-logic solutions are more likely to benefit from the wider bus.

The third protocol supported is the SPI mode of the SD Card protocol. It is distinct from the 1-bit and 4-bit protocols in that the protocol operates over a generic and well-known bus interface, Serial Peripheral Interface (SPI). SPI is a synchronous serial protocol that is extremely popular for interfacing peripheral devices with microcontrollers. Most modern microcontrollers, including the MSP430, support SPI natively at relatively high

data rates. The SPI communications mode supports only a subset of the full SD Card protocol. However, most of the unsupported command sets are simply not needed in SPI mode. A fully-functional SD Card implementation can be realized using only SPI.

This flexibility of electrical interfaces is a significant advantage to a designer. A designer may opt for a fast parallel interface, or depending on the application, may prefer a slower implementation using SPI. Due to the popularity of the SPI protocol and its efficient implementation on the MSP430, this application note covers only the SPI mode of the SD Card protocol. Minor differences in the initialization sequence exist between the two major modes. For more information, the interested reader should consult the full SD Card Specification, available from the SD Card Association [3].

Figure 2 shows the SD Card electrical interface for SPI mode. The external pullups shown are required by the SD protocol, and must be present even for the unused data pins. Note that the $\overline{SS}$ pin must remain selected over a period of many octets sent via SPI, and as such, cannot be used with the automatic slave-selection pin as implemented by the MSP430 and requires handling in software.

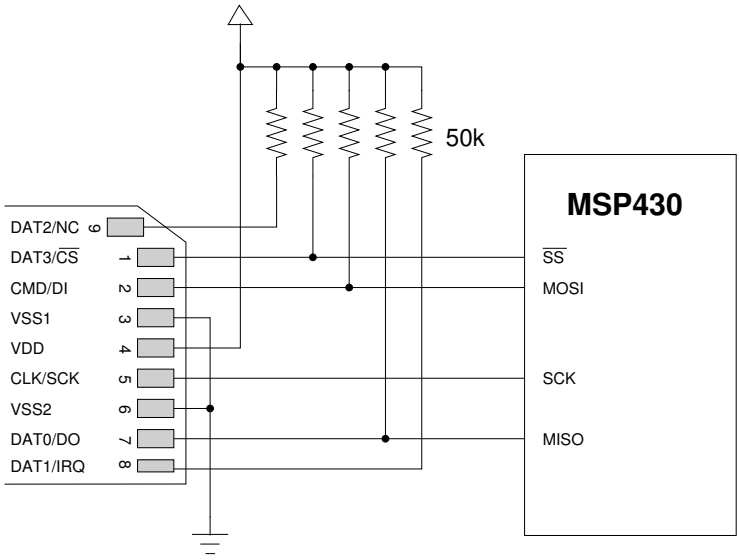

Figure 2: SD Card Schematic – SPI Mode [2].

Protocol

The SD Card protocol described herein is the *SPI* mode of the SD Card Protocol. The initialization sequence, frame format, response format, and command identifiers all differ for SD 4-bit and SD 1-bit modes.

The SD protocol is a simple command-response protocol. All commands are initiated by the master. The SD card responds to the command with a response frame and then, depending on the command, may be followed by a data token indicating the beginning of a bulk data transfer or an error condition.

SD commands are listed in the form "CMDXX" or "ACMDXX," where the tags "CMD" and "ACMD" refer to general commands and application-specific commands, respectively, and "XX" is the command number. SD commands are issued to the card in a packed command frame, a 6-byte structure sent over the SPI port. The command frame always starts with "01" followed by the 6-bit command number. Next the 4-byte argument is

sent, MSB first. The 7-bit CRC with a final stop bit '1' is sent last. All bytes of the command frame are sent over the MOSI pin MSb first. Table 2 shows the command frame format. Note that the CRC is optional in SPI mode and by default CRC checking is disabled.

First Byte			Bytes 2-5	Last Byte	
0	1	Command	Argument (MSB First)	CRC	1

Table 2: SD Command Format [2].

The SD card responds to each command frame with a response. Every command has an expected response type. The type of response used for a particular command depends only on the command number, not on the content of the frame. Three response types are defined for SPI mode: R1, R2, and R3. The bitfield designations are described in Tables 3, 4, and 5, respectively.

Byte	Bit	Meaning
1	7	Start Bit, Always 0
	6	Parameter Error
	5	Address Error
	4	Erase Sequence Error
	3	CRC Error
	2	Illegal Command
	1	Erase Reset
	0	In Idle State

Table 3: Response type R1 [2].

Byte	Bit	Meaning
1	7	Start Bit, Always 0
	6	Parameter Error
	5	Address Error
	4	Erase Sequence Error
	3	CRC Error
	2	Illegal Command
	1	Erase Reset
	0	In Idle State
2	7	Out of Range, CSD Overwrite
	6	Erase Parameter
	5	Write Protect Violation
	4	Card ECC Failed
	3	Card Controller Error
	2	Unspecified Error
	1	Write Protect Erase Skip, Lock/Unlock Failed
	0	Card Locked

Table 4: Response type R2 [2].

Byte	Bit	Meaning
1	7	Start Bit, Always 0
	6	Parameter Error
	5	Address Error
	4	Erase Sequence Error
	3	CRC Error
	2	Illegal Command
	1	Erase Reset
	0	In Idle State
2-5	All	Operating Condition Register, MSB First

Table 5: Response type R3 [2].

Bulk data transfers provide a mechanism to efficiently transfer large amounts of data to and from the SD Card. The normal command-response structure only allows for the exchange of small amounts of data in fixed sizes, while bulk data may be of any arbitrary size.

When a bulk data command is issued to the card, the card responds normally with one of the three standard response types. Then the bulk transfer starts with a data token, followed by the bulk data itself, and completes with a 16-bit CRC [2].

For a block read or write, the block transfer is preceded by a start block token, a constant "11111110." This is followed by a block of data (typically 512 bytes), and then followed by a 16-bit CRC [2]. The CRC is not calculated and is ignored in this implementation.

After every card write, the card will return a 1-byte token indicating the status of the operation. The response token is "XXX0AAA1," where XXX are don't cares and AAA indicates the status, if any. "010" indicates the data was accepted. "101" indicates the data was rejected because of a CRC error. "110" indicates the data was rejected because of a write error [2].

Finally, if a block read fails, the card will return a 1-byte error token. The format of this token is shown in Table 6.

Bit	Meaning
7	Always '0'
6	Always '0'
5	Always '0'
4	Card Locked
3	Out of Range
2	Card ECC Failed
1	Card Controller Error
0	Unspecified Error

Table 6: Read Error Token [2].

Relevant command numbers are listed in table 7. This does not represent the full list of commands. Only those critical to this implementation are listed.

Command	Argument	Type	Description
CMD0	None	R1	Tell the card to reset and enter its idle state.
CMD16	32-bit Block Length	R1	Select the block length.
CMD17	32-bit Block Address	R1	Read a single block.
CMD24	32-bit Block Address	R1	Write a single block.
CMD55	None	R1	Next command will be application-specific (ACMDXX).
CMD58	None	R3	Read OCR (Operating Conditions Register).
ACMD41	None	R1	Initialize the card.

Table 7: Important SD Commands [2].

Implementation

The SD card controller is implemented as a simple low-level block driver with API calls for initialization, block reads, block writes, and a minimal set of status functions.

Command and Response

The function `sd_send_command()` forms the basis for all other SD functions. The function makes use of the MSP430's onboard SPI controller to send and receive bytes through the SPI port. The $\overline{\text{SS}}$ (slave select) pin is toggled directly through software, as the SD SPI protocol requires that $\overline{\text{SS}}$ remain asserted (0) throughout the command frame.

Functionally, `sd_send_command()` asserts $\overline{\text{SS}}$, packs the command frame, and then sends the command frame, argument, and a constant CRC 0x95. The CRC 0x95 is always used since it is the correct CRC for a CMD0 with a zero argument. This has to do with the initialization sequence, and will be explained further in the section "Card Initialization."

Then, based on the expected response type, the function will read a number of bytes into a buffer and return an error if a response is not received within a timeout interval. Before returning, $\overline{\text{SS}}$ is deasserted.

Card Initialization

SD cards require a specific initialization sequence. Parts of the initialization sequence are historical, and other parts are required for backward and forward compatibility. MMC and SD are not substantially different; the primary difference from a software point of view is in the initialization sequence.

Card initialization starts by setting the SPI clock to 400kHz. This is required for compatibility across a wide range of MMC and SD Cards.

Next, at least 74 clocks must be issued by the master before any attempt is made to communicate with the card. This allows the card to initialize any internal state registers before card initialization proceeds.

Next, the card is reset by issuing the command CMD0 while holding the $\overline{\text{SS}}$ pin low. This both resets the card and instructs it to enter SPI mode. Note that while the CRC, in general, is ignored in SPI mode, the very first command must be followed by a valid CRC, since the card is not yet in SPI mode. The CRC byte for a CMD0 command with a zero argument is a constant 0x95 [1]. For simplicity, this CRC byte is always sent with every command.

Next, the card is continuously polled with the commands CMD55 and ACMD41 until the idle bit becomes clear, indicating that the card is fully initialized and ready to respond to general commands.

Next, the command CMD58 is used to determine if the card supports the processor's operating voltage. CMD58 returns a bitfield containing the allowed operating voltage ranges, typically between 2.7V and 3.6V. It is assumed that the MSP430 is using a voltage supply of 3.3V.

Finally, the SPI clock is set to the maximum rate allowed.

The initialization sequence is implemented by the function `sd_initialize()`. This function is specific to SD cards in SPI mode, and will not properly initialize an MMC card or a card intended to be used in the 1-bit or 4-bit SD Bus mode. The initialization sequence is characterized in the flowchart in Figure 3.

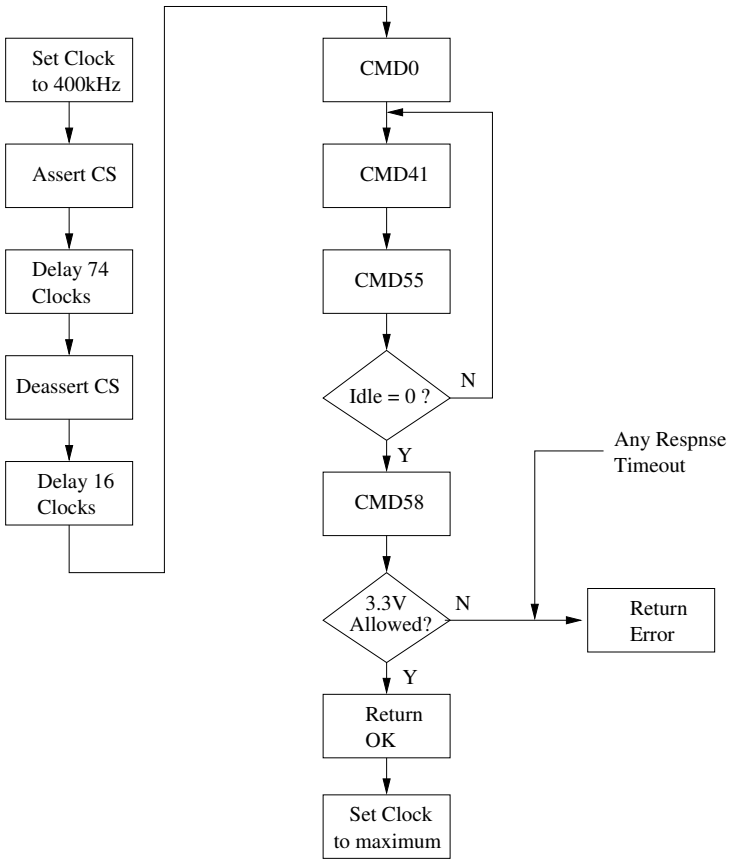

Figure 3: SD Card Initialization Sequence.

Block Read

The block read command is a bulk data command. The command response is followed by a delay, then followed by a "start of block" token, and then followed by the actual block itself.

In order to make the most efficient use of resources and enable fast block transfers, the block read function uses the DMA (Direct Memory Access) controller on the MSP430. First, the command is sent and the response is

received. Then, the function waits until the start token is received. When it is received, the function starts a DMA transfer.

Since SPI requires that a byte be sent for a byte to be received, two DMA units are used to complete the transfer.

DMA0 is triggered by a UART receive. The source for the DMA transfer is the USART receive buffer, U0RXBUF. The source is set to byte-wide, no increment. The destination for the DMA transfer is the data buffer. The destination is set to byte-wide, with an increment. The count is fixed at 512, the default block size for a typical SD card.

DMA1 is also triggered by a UART receive. The source for this register is a constant 0xFF (the idle bus condition). The output is the USART transmit buffer, U0TXBUF.

DMA priorities ensure that a byte will be received before a new 0xFF (idle byte) is sent. Since both DMA units use the same trigger, DMA0 will always be serviced before DMA1.

Finally, the receive and transmit interrupt flags are reset and the entire block transfer is triggered by manually sending a single idle byte.

The function `sd_read_block()` implements the block read. The function will return immediately and normal program execution can continue while the block transfer finishes. `sd_wait_notbusy()` can be used to synchronously wait for any pending block transfers to finish.

Block Write

The block write is similar to the block read function in that it uses a DMA transfer and also starts with a data token. However, since no bytes need to be received during the block transfer, the block transfer only requires one DMA trigger.

DMA0 is triggered by a UART send. The destination for the DMA transfer is the USART receive buffer, U0RXBUF. The destination is set to byte-wide, no increment. The source for the DMA transfer is the data buffer. The source is set to byte-wide, with an increment. The count is fixed at 512, the default block size for a typical SD card.

Finally, the receive and transmit interrupt flags are reset and the entire block transfer is triggered by manually sending a single idle byte.

The function `sd_write_block()` implements the block write. The function will return immediately and normal program execution can continue while the block transfer finishes. `sd_wait_notbusy()` can be used to synchronously wait for any pending block transfers to finish.

Summary and Conclusions

SD Cards offer a cost-effective way to store large amounts of data in a removable memory storage device. The simplicity of the SD Card protocol and the flexibility in interfacing with these devices makes them ideal for use with small microcontrollers.

Combined with the low-cost, low-power TI MSP430 and its advanced features like onboard DMA and SPI, a fast and low-overhead complete data logging solution can be implemented quickly and inexpensively. Additional application-level support for a filesystem such as FAT16 can extend the usefulness of this solution even further.