Altera University Program
Secure Data Card IP Core

SOPC Builder IP Core

1 Core Overview

A Secure Data (SD) card is a data storage device, which is often used in digital cameras to store images. An SD card is portable, which allows the data stored on it to be transferred to other devices. Altera DE1, DE2, and DE3 boards have an SD card port. It allows an SD card to be connected to an FPGA-based design on these boards, facilitating access to potentially large amounts of data.

The Altera University Program (UP) SD Card IP Core is a hardware circuit that enables the use of an SD card on the Altera DE1, DE2, and DE3 boards. When it is included in a design and connected to the SD card port, the core will detect any SD card connected to the SD card port, and allow a user circuit to easily access data stored there.

The core has been designed to be used in an SOPC Builder-based system. When that system also includes a Nios II soft-core processor, data on an SD card can be accessed by programs running on the Nios II processor. Programmers can use a simple programming model to access the data stored on an SD card.

In the following sections, we describe the Altera UP SD Card IP Core in more detail. We show how to instantiate the IP core using the SOPC Builder tool, and discuss the software programming model.

2 Functional Description

The University Program SD Card IP Core is designed to use the SD Port on the DE1, DE2 and DE3 boards to access data on an SD card. The core functions as an interface between the SD card and a system created in the SOPC Builder tool.

A high-level block diagram of the core is shown in Figure 1. The signals on the left-hand side of the figure connect to the Avalon interconnect. Read and write requests received through the Avalon interconnect are interpreted as either *command* or *data* requests by the Avalon Interface Finite State Machine (FSM). Command requests are used to configure the SD card and specify locations on the SD card we wish to access. The data requests are used to access to the raw data stored on the SD card. Once the FSM determines the type of a request issued, it enables the SD Card Interface module. The SD Card Interface module processes the request by communicating with the SD card using serial communication protocol, and returns the result of the request to the Avalon Interface FSM. The Avalon Interface FSM in turn sends the result of the request out through the Avalon interconnect and signals that it has completed operation.

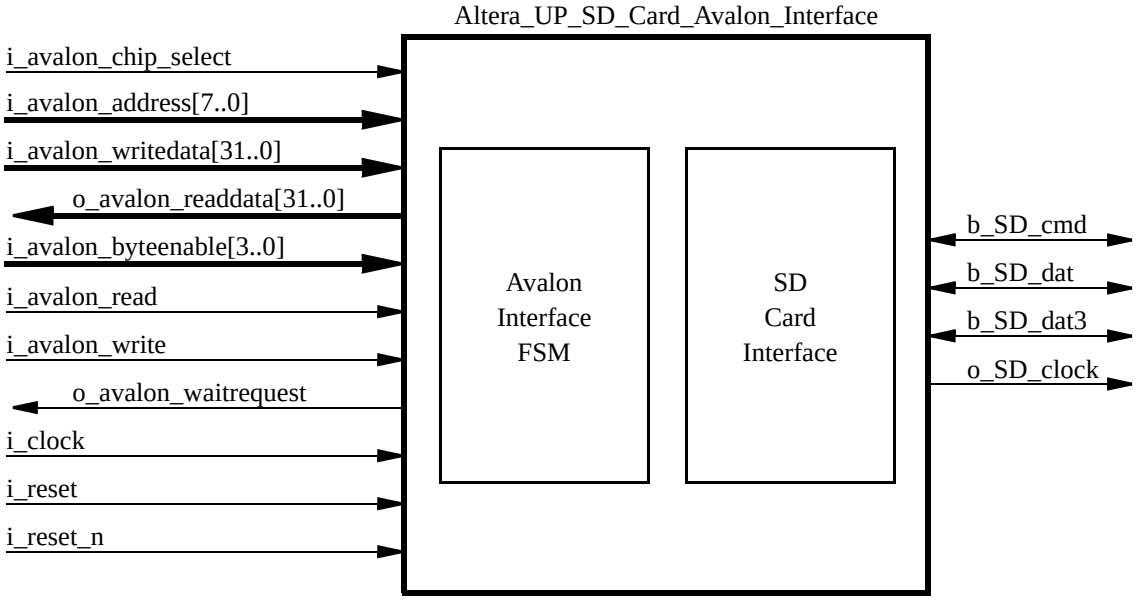

Figure 1. High-level diagram of the Altera UP SD Card IP Core.

3 Instantiating the Core in SOPC Builder

To include the Altera UP SD Card IP Core in an SOPC Builder-based design, users need to instantiate the core in their SOPC Builder design. To do this, first locate the "SD Card Interface" core under University Program > Memory. Then add the core to your design and assign an address range to it. The address range occupied by the SD Card core spans 1024 bytes of addressable space. The address range should begin at an address whose ten least-significant bits are zeros. For example, a starting address of 0x00000400 is acceptable.

Once the core is included in your design, it will be necessary to connect SD card ports to appropriate pins on the FPGA device. Table 1 shows the ports and their associated pins on the DE1, with Altera Cyclone II FPGA device (EP2C20F484C7), on the DE2 board, with Altera Cyclone II FPGA device (EP2C35F672C6), and on the DE3 board with Altera Stratix III device (EP3SL150F1152C2). Finally, it is important to set T_{co} and T_{su} constraints for the SD card ports. Both parameters should be set to no more than 10ns. Should the parameters be omitted during design compilation, it is possible that the core may malfunction.

Table 1. Pin Assignments for SD card port on DE1, DE2 and DE3 boards.

Pin Name	DE1	DE2	DE3
b_SD_cmd	PIN_Y20	PIN_Y21	PIN_R10
b_SD_dat	PIN_W20	PIN_AD24	PIN_P7
b_SD_dat3	PIN_U20	PIN_AC23	'Z'
o_SD_clock	PIN_V20	PIN_AD25	PIN_P8

Note: The b_SD_DAT3 pin is not used on the DE3 board. The IP core will still have an b_SD_DAT3 bidirectional port, which should be set to high impedance (Z).

4 Software Programming Model

When the Altera University Program SD Card IP Core is included in an SOPC Builder design with a Nios II soft-core processor, the core can be accessed and controlled from software. Software programs can be written to either communicate with the SD card directly, using memory-mapped registers and a memory-mapped buffer, or by using a Hardware Abstraction Layer (HAL) device driver that makes an SD card appear as a 16-bit File Allocation Table (FAT16)-based portable drive. In the following sections we describe how to use these two programming models.

4.1 Direct SD Card Communication

The memory-mapped registers/buffer in the Altera UP SD Card Interface IP Core can be used to exchange information between your system and an SD card. The memory-mapped registers allow a program running on the Nios II soft-core processor to read the status of the SD Card Interface as well as send commands to it. The commands include reading, writing and erasing a block of data. When a command to read a block of data is issued, the core reads a 512-byte block of data into a local memory buffer. Once the data is stored in the buffer, the buffer can be read and written to using memory reads/writes from a software program.

4.1.1 Memory-Mapped Registers

The location in memory of the memory-mapped registers and the data buffer, relative to the starting address as indicated in the SOPC Builder project created by a user, are listed in Table 2.

Table 2. SD Card IP Core Register Map

Offset in bytes	Size in bytes	Register Name	R/W	Register Description
0	512	RXTX_BUFFER	R/W	Data buffer for incoming and outgoing data
512	16	CID	R	Card Identification Number Register
528	16	CSD	R	Card Specific Data Register
544	4	OCR	R	Operating Conditions Register
548	4	SR	R	SD Card Status Register
552	2	RCA	R	Relative Card Address Register
556	4	CMD_ARG	R/W	Command Argument Register
560	2	CMD	R/W	Command Register
564	2	ASR	R	Auxiliary Status Register
568	2	RR1	R	Response R1

Registers listed in Table 2 are accessible by reading and/or writing data to the specified memory locations. Registers CID, CSD, OCR, SR, and RCA are described in the SD Card Physical Layer Specification document. The meaning of bits in these registers is described there. The CMD_ARG, CMD, ASR, and RR1 registers, as well as the RXTX_BUFFER buffer, are specific to the Altera University Program SD Card Core. In the following section we describe how to use these registers and the buffer to communicate with the SD card.

4.1.2 Using Memory-Mapped Registers to Communicate with an SD Card

The Altera UP SD Card IP Core abstracts the low-level SD card communication protocol using memory-mapped registers. It can transfer data to and from an SD card requiring only that users wait for each transaction to be completed. To facilitate this level of abstraction, the core uses three registers and a memory buffer.

The first register is the Auxiliary Status Register, or ASR. This register stores status information for the core. The meaning of each bit is as follows:

- bit 0 indicates if the last command sent to the core was valid
- bit 1 indicates if an SD Card is present in the SD card socket
- bit 2 indicates if the most recently sent command is still in progress
- bit 3 indicates if the current state of the SD card Status Register is valid
- bit 4 indicates if the last command completed due to a time out
- bit 5 indicates if the most recently received data contains errors

A software application can poll this register to determine the state of the core, without interrupting the operation being performed by the core. For example, an application can continuously poll bit 1 of the ASR to wait until an SD card is inserted into an SD card socket. While the bit is being polled, the core will continue operating, waiting for an SD card to be inserted into the SD card socket. Once a card is detected in the slot, the core will initialize the card, and if successful, the core will set bit 1 of the ASR to indicate that an SD card is ready for access.

Once the card is initialized by the core, it can be accessed by using various commands. Although the Altera UP SD Card IP Core supports a wide array of SD card functions (see Appendix A), the most frequently used commands are READ_BLOCK and WRITE_BLOCK.

To execute the READ_BLOCK command, write the address to read from into the Command Argument Register (CMD_ARG). For example, to read from the SD Card starting at address 0x00001000, write 0x00001000 to CMR_ARG register. Then, write the READ_BLOCK command ID (0x0011) to the Command (CMD) register. This sequence of events causes the SD Card core to read data from the SD Card, starting at the address 0x00001000. When the command completes execution, the requested data will be accessible via the RXTX_BUFFER. Please note that the data is read in 512 byte blocks, thus it is only necessary to issue a read command for the given block once. Once the block is read, the RXTX_BUFFER can be accessed to read data from the given block. Also note that the address specified in the CMD_ARG register must be an integer multiple of 512 bytes.

As an example, consider the example code in Figure 2. In this example, we first wait for the SD card to be connected to the SD card socket. Once a card is detected, we proceed to read 11^{th} sector on the SD card. The 11^{th} sector begins on byte 5120 and ends on byte 5631. Note that when the command to read data from the SD card has been sent, the program waits in a loop. This is because the operation may take some time and the data will not be available immediately. It is therefore necessary to wait until ASR register indicates that the read operation has been completed.

```
#define READ_BLOCK 17
int main(void) {
    int *command_argument_register = ((int *)(0x0000122C));
    short int *command_register = ((short int *)(0x00001230));
    short int *aux_status_register = ((short int *)(0x00001234));
    short int status;

    /* Wait for the SD Card to be connected to the SD Card Port. */
    do {
        status = *aux_status_register;
    } while ((status & 0x02) == 0);

    /* Read 11th sector on the card */
    *command_argument_register = (10) * 512;
    *command_register = READ_BLOCK;

    /* Wait until the operation completes. */
    do {
        status = *aux_status_register;
    } while ((status & 0x04)!=0);
}
```

Figure 2. Example of reading a block of data from an SD card.

Executing WRITE_BLOCK is performed in the same manner as executing READ_BLOCK. However, before the WRITE_BLOCK is executed, the RXTX_BUFFER should be filled with 512 bytes of data to be written to the SD card. Once the buffer contains the desired data, write the destination address to the CMD_ARG register (a multiple of 512 bytes as for the read command), and then write WRITE_BLOCK command ID (0x0018) to the CMD register.

IMPORTANT: Please note that an SD card is a flash memory device, and as such writing to it takes longer than reading data from it. Also, each 512 block of data on an SD card can only be written a limited number of times (depending on the SD card used, this number varies between 1000 and 100000 times), thus users should take care to write to the SD card only when necessary.

When using both the read and the write commands, the RR1 register will contain the response to a read/write request. In particular, the following bits of RR1 are significant:

- bit 30 is high if address or block length parameters was out of range when requesting a read/write operation
- bit 29 is high if the address was misaligned
- bit 28 is high if an error in a sequence of erase commands occurred
- bit 27 is high if the CRC check of the last command failed

- bit 26 is high if the specified command was illegal
- bit 25 is high if an erase sequence has been interrupted by another command
- bit 24 is high if the card is currently running the initialization procedure

Other commands supported by the Altera University Program SD Card IP Core are listed in Appendix A.

4.2 Hardware Abstraction Layer Device Driver

The Hardware Abstraction Layer (HAL) device driver designed for the Altera University Program SD Card IP Core provides an easy way to access data stored on an SD card. The driver functions as a File Allocation Table (FAT) reader/writer, allowing users to access data on the SD card that has been saved in FAT16 format. There are several versions of FAT format, including FAT12, FAT16 and FAT32 (with long names), however the current version of the driver only supports FAT16.

4.2.1 Formatting the SD Card

In a FAT16 file format, data is saved into bins called clusters. Each cluster has an ID number that can range from 2 to 65520. A file is created by filling a cluster with data, and setting a flag to indicate in which cluster the next set of data for the file is located. This is a reasonably simple scheme and has been in use for a long time. The Altera UP SD Card IP Core device driver works with an SD card that is formated such that it contains at least 4087 clusters and no more than 65520 clusters, thereby forcing each cluster to have a 16-bit ID (FAT16 format). An SD card can be formatted to match these specifications in Microsoft Windows using the format command. For example, if in Windows your SD card is specified to be in drive H, then:

```
format H: /FS:FAT /A:2048 /V:SDCARD
```

will format the SD card with a FAT such that each cluster contains 2048 bytes of data. The format command will show the following information when a 16MB SD card is formatted using the above command:

```
The type of the file system is FAT.
Verifying 14M
Initializing the File Allocation Table (FAT)...
Format complete.

    14,829,568 bytes total disk space.
    14,829,568 bytes available on disk.

        2,048 bytes in each allocation unit.
        7,241 allocation units available on disk.

            16 bits in each FAT entry.
```

Notice that the format program reports that each FAT entry takes 16 bits, which is exactly the configuration our device driver works with. Once the SD card is formatted correctly, users can store files on the SD card.

4.2.2 Using HAL Device Driver

To use HAL Device Driver subroutines include the *Altera_UP_SD_Card_Avalon_Interface.h* file in your program and recompile it. If you are using the Altera Monitor Program to compile your project, you have to execute the Regenerate Device Drivers (BSP) command from the Actions menu before recompiling your project.

To use the SD card driver, users must first initialize the driver by calling the following function:

```
alt_up_sd_card_dev* alt_up_sd_card_open_dev(const char* name)
```

This function takes as input the instance name of the Altera UP SD Card IP Core component in the SOPC Builder system, preceded by a "/dev/" string. For example, if the core instance name is called "Interface", then the parameter to the above function should be "/dev/Interface". If successful, the function returns a non-NULL pointer to a data structure, which contains a base address for the SD Card IP core in the specified system. Once the driver is initialized, other functions in the driver become available.

For example, consider the program in Figure 3. It initializes the SD card device driver, and then continuously checks for the presence of an SD card the SD card socket. If an SD card is detected, the application checks if it contains a FAT16 file system. If the card is removed from the SD card slot, the application informs the user that the card is no longer present in the SD card socket.

The code in Figure 3 is a simple example of how to use Altera University Program SD Card IP Core HAL device driver. A complete list of subroutines available in the device driver is shown in Appendix B.

5 Summary

This document described the Altera University Program SD Card IP Core for Altera DE2 board. The core has been implemented based on SD Card Physical Layer Specification document, version 1.10, dated April 3, 2006. A demonstration program of how this core can be used, called *Altera University Program SD Card Demo*, can be found on Altera University Program Website.

```
#include <stdio.h>
#include <altera_up_sd_card_avalon_interface.h>
int main(void) {
    alt_up_sd_card_dev *device_reference = NULL;
    int connected = 0;

    device_reference = alt_up_sd_card_open_dev("/dev/Interface");
    if (device_reference != NULL) {
        while(1) {
            if ((connected == 0) && (alt_up_sd_card_is_Present())) {
                printf("Card connected.\n");
                if (alt_up_sd_card_is_FAT16()) {
                    printf("FAT16 file system detected.\n");
                } else {
                    printf("Unknown file system.\n");
                }
                connected = 1;
            } else if ((connected == 1) && (alt_up_sd_card_is_Present() == false)) {
                printf("Card disconnected.\n");
                connected = 0;
            }
        }
    }
    return 0;
}
```

Figure 3. Using HAL device drivers to check if an SD card is present in the SD card slot.

6 Appendix A - Supported SD Card Instructions

Table 3 lists instructions that can be executed by the SD Card using the Altera University Program SD Card IP Core.

Table 3. Supported SD Card Instructions

Name	Command ID	Argument	Description
SEND_ALL_CID	0x0002	None	Causes the SD card to send its CID number. This ID can be read using the CID memory mapped register.
SEND_RCA	0x0003	None	Causes the SD card to send its RCA number.
SET_DSR	0x0004	Top 16 bits of CMD_ARG must contain DSR.	Programs the SD Card's DSR register.
SEND_CSD	0x0009	Top 16 bits of CMD_ARG must contain RCA. This can be accomplished by using command ID 0x0049 instead.	Causes the SD Card to send its Card Specific Data register to the Core. This data can be accessed by reading the memory mapped CSD register.
SEND_CID	0x000A	Top 16 bits of CMD_ARG must contain RCA. This can be accomplished by using command ID 0x004A instead.	Causes the SD Card to send its Card Identification Number to the Core. This data can be accessed by reading the memory mapped CID register.
SEND_STATUS	0x000D	Top 16 bits of CMD_ARG must contain RCA. This can be accomplished by using command ID 0x004D instead.	Causes the SD Card to send its 32-bit status register to the Core. This register can be accessed by reading the memory mapped SR register.
READ_BLOCK	0x0011	Must contain a valid address that is a multiple of 512.	Reads a 512 byte block of data from the SD card at the specified address into the RXTX_BUFFER.

Name	Command ID	Argument	Description
WRITE_BLOCK	0x0018	Must contain a valid address that is a multiple of 512.	Write a 512 byte block of data from the RXTX_BUFFER to the SD card at the specified address.
SET_WRITE_PROTECT	0x001C	Must contain a valid address that is a multiple of 512.	Sets a flag that designates the block to be write-protected.
CLR_WRITE_PROTECT	0x001D	Must contain a valid address that is a multiple of 512.	Clears a flag that designates the block to be write-protected.
ERASE_BLOCK_START	0x001E	Must contain a valid address that is a multiple of 512.	Specifies the block address where earsing should begin.
ERASE_BLOCK_END	0x001F	Must contain a valid address that is a multiple of 512.	Specifies the last block to be erased.
ERASE	0x0026	None	Erases the previously selected array of blocks on the SD card.
APP_CMD	0x0038	Top 16 bits should contain RCA. This can be accomplished by using command code 0x0078 instead.	Allows the next instruction to be executed to be an Application Specific Instruction, as defined by the SD Card Physical Layer Specification 1.10 document.

The above list contains the main set of functions accepted by the Altera University Program SD Card IP Core. In addition to the above functions, the SD Card Physical Layer Specification document lists application specific commands that the card can accept. These commands are similar to the basic commands, however they must be preceeded by APP_CMD command. The list of supported Application Specific commands is given in Table 4.

Table 4. Supported SD Card Application Specific Instructions

Name	Command ID	Argument	Description
BLK_ERASE_COUNT	0x0017	Bottom 23 bits specify the number of blocks to erase when writing to an SD card.	Set the number of blocks to be erased before writing. By default this value is set to 1.
SEND_OCR	0x0029	OCR	Causes an SD card to send its operation conditions register to the core. The register contents can then be accessed by reading the OCR memory-mapped register. Note that this command is executed when an SD card is initialized and the contents of the register are available once the core indicates an SD card is present in the SD card slot.
SEND_SCR	0x0033	None	Causes the SD card to send its Configuration Register to the core. The contents of the register can be read by accessing the SCR memory-mapped register. Note that this command is executed when an SD card is initialized and the contents of the register are available once the core indicates an SD card is present in the SD card slot.

7 Appendix B - Hardware Abstraction Layer Device Driver Subroutines

The following list shows functions available in the Altera University Program SD Card IP Core HAL device driver:

Prototype:	alt_up_sd_card_dev* alt_up_sd_card_open_dev(const char *name)
Inputs:	const char *name - the instance name of SD Card IP Core in the SOPC Builder system
Outputs:	alt_up_sd_card_dev* - a pointer to a structure holding a *base* field. The *base* field holds the address of the SD Card IP Core in the SOPC Buidler system.
Description:	Initializes the SD Card IP Core HAL device driver. Returns NULL, when the specified device name does not exist in the system.

Prototype:	short int alt_up_sd_card_fopen(char *name, bool create)
Inputs:	char *name - name of the file to open, relative to root directory bool create - set to **true** if the file should be created if it is not found
Outputs:	short int - A handle to the opened file. A negative result indicates an error as follows: -1 means that the file could not be opened, and -2 means the file is already open.
Description:	Opens a file for use in your application.

Prototype:	short int alt_up_sd_card_find_first(char *directory_to_search_through, char *file_name);
Inputs:	char *directory_to_search_through - name of the directory to search through, relative to root directory char *file_name - a pointer to an array to store the name of the file found by this function
Outputs:	short int - result of the operation. 0 means success, 1 means an invalid directory, 2 means no card is present or the card does not contain a FAT16 partition, and -1 means that the directory has been scanned and no files were found.
Description:	Looks for the first file in a given directory.

Prototype:	short int alt_up_sd_card_find_next(char *file_name);
Inputs:	char *file_name - a pointer to an array to store the name of the file found by this function
Outputs:	short int - result of the operation. 0 means success, 1 means an invalid directory, 2 means no card is present or the card does not contain a FAT16 partition, 3 means that find_first must be called first, and -1 means that the directory has been scanned and no files were found.
Description:	Looks for the next file in a directory specified in the last call to the alt_up_sd_card_find_first subroutine.

Prototype:	short int alt_up_sd_card_get_attributes(short int file_handle)
Inputs:	short int file_handle - handle to a file as returned by the alt_up_sd_card_fopen
Outputs:	short int - File attributes when successful, -1 otherwise.
Description:	Returns the attributes of the specified file.

Prototype:	void alt_up_sd_card_set_attributes(short int file_handle, short int attributes)
Inputs:	short int file_handle - handle to a file as returned by the alt_up_sd_card_fopen short int attributes - file attributes as specified by FAT16 file system
Outputs:	short int - File attributes when successful, -1 otherwise.
Description:	Sets attributes for the specified file.

Prototype:	short int alt_up_sd_card_read(short int file_handle)
Inputs:	short int file_handle - handle to a file as returned by the alt_up_sd_card_fopen
Outputs:	short int - a byte of data when successful, or a negative value when failure occurs. -1 means that the file handle was invalid, and -2 indicates inability to read from the SD card.
Description:	Reads a byte of data from a given file at current position in the file. The position in the file is incremented when data is read.

Prototype:	bool alt_up_sd_card_write(short int file_handle, unsigned char byte_of_data)
Inputs:	short int file_handle - handle to a file as returned by the alt_up_sd_card_fopen unsigned char byte_of_data - a byte of data to be written
Outputs:	bool - **true** when successful and **false** when unsuccessful
Description:	Writes a byte of data at the current position in the file. The position in the file is incremented when data is written.

Prototype:	bool alt_up_sd_card_fclose(short int file_handle)
Inputs:	short int file_handle - handle to a file as returned by the alt_up_sd_card_fopen
Outputs:	bool - **true** when successful and **false** when unsuccessful
Description:	Closes a previously opened file and invalidates the given file_handle.

Prototype:	bool alt_up_sd_card_is_Present(void)
Inputs:	None
Outputs:	bool - **true** when an SD card is present in the SD card socket, and **false** otherwise
Description:	Checks if an SD card is present in the SD card socket

Prototype:	bool alt_up_sd_card_is_FAT16(void)
Inputs:	None
Outputs:	bool - **true** when an SD card contains FAT16 data, and **false** otherwise
Description:	Checks if an SD card contains a FAT16 partition.