SDCard
1.0

Features and Overview

- Supports SD, miniSD, microSD/TransFlash, MMC, RS-MMC/MMCmobile, and MMCplus.
- Handles PC FAT16/32, DOS, and Windows files with short filenames (DOS 8.3 format).
- Opens multiple files for read and write operations.
- Supports multiple file random access.
- Allows PSoC to access 2 Gb of flash storage space.

The SDCard Component allows you to access PC compatible files on six different flash card form factors without the need to know the "nuts and bolts" of either file access or the flash card interface.

The SDCard Component allows basic operation with as few as four PSoC pins. Depending upon the card type and card socket, you can use additional pins to support write protect, card insert, and others.

This component allows you to access SD and MMC cards using a simple C interface. There is no need to know how either the SPI bus or the SD/MMC command set works as long as you use the SDCard APIs.

Use any SD or MMC card with this user module as long as it meets these requirements:

- The operating voltage range falls within the voltage being used in the design.
- You use the SPI data mode to address the card.
- The card meets the specifications found on the sdcard.org or mmca.org web sites.
- In order to work with PC compatible cards, you must also follow these additional requirements:
- The card is formatted with a Windows/DOS compatible FAT16 (or optional FAT32) file structure.
- The card is formatted as a hard disk drive using a partition table in the first sector.
- The files to read are in the root directory only. Subdirectories are not currently supported.

Note: In general, cards less than 32 MB are formatted as FAT12. Cards 32 MB and up (with a maximum FAT16 size of 16 GB) are formatted as FAT16. However, there are exceptions to any rule. Windows or another card utility may format the cards in a different format. For instance, larger memory cards may be formatted as FAT32. Most cards can be formatted as FAT16. Check the software for formatting options.

Figure 1: Interfacing an SD Card to a PSoC Device Operating at 3.3 V

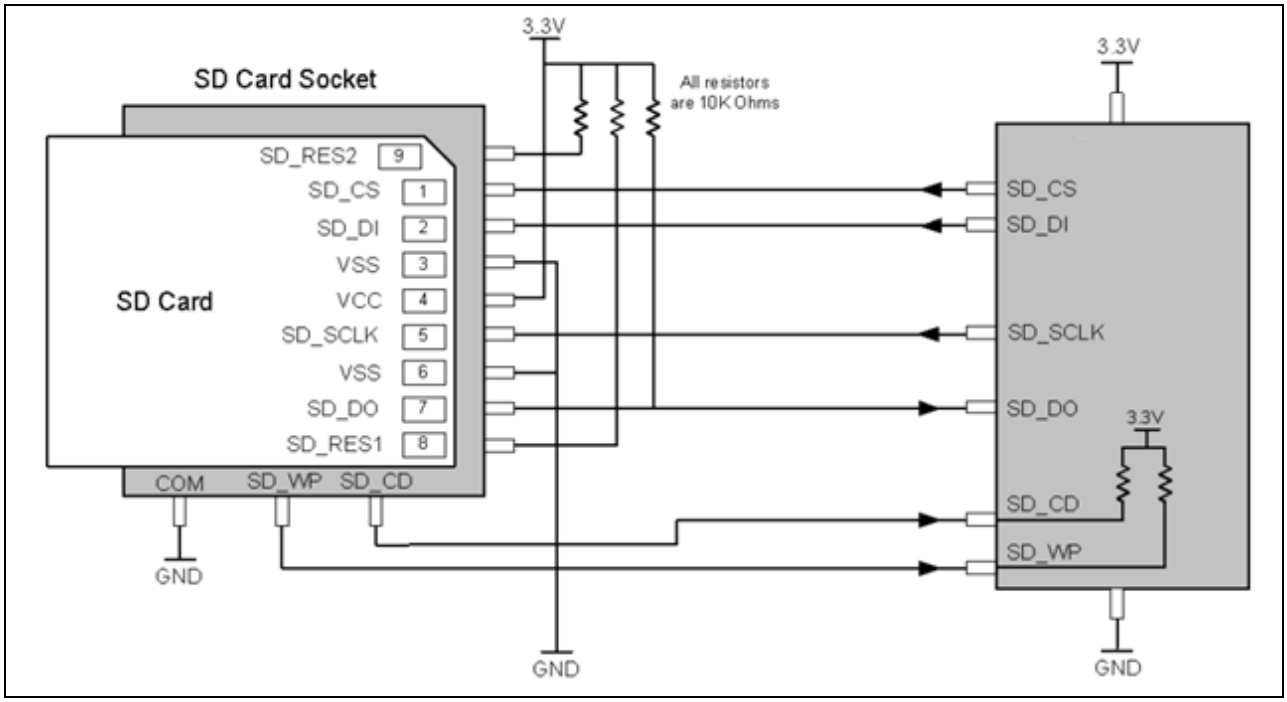


Figure 2: Interfacing an SD Card to a PSoC Device Operating at 5 V

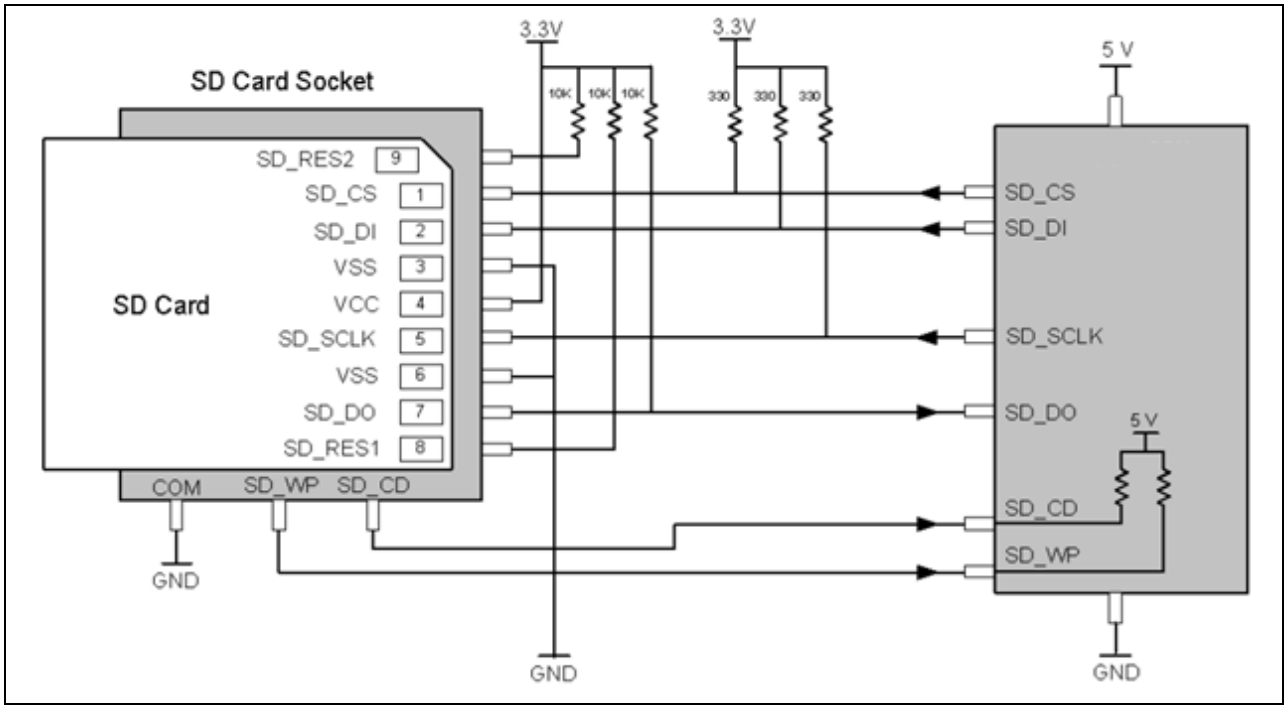
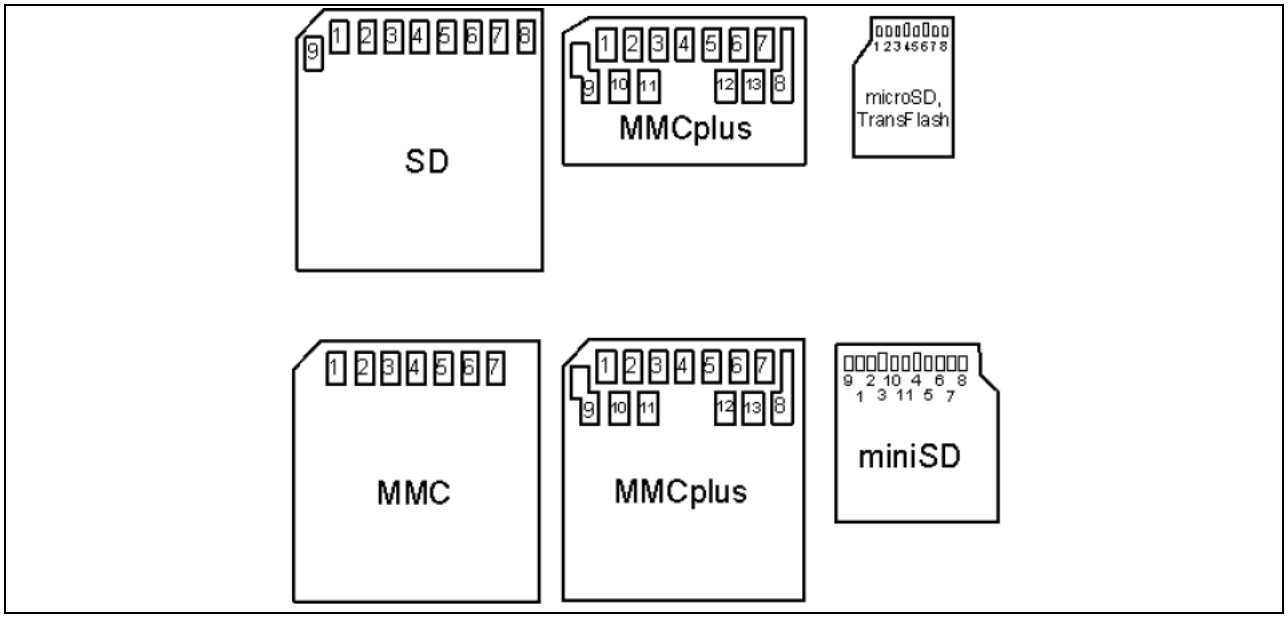


Table 1: PSoC to Flash Card Pins

Component Pin	Description	SD	miniSD	microSD, TransFlash	MMC	RS_MMS, MMC Plus, MMCmobile
SD_CS	Card Select	1	1	2	1	1
SD_MOSI	PSoC-to-Card Commands and Data	2	2	3	2	2
SD_MISO	Card-to-PSoC Data	7	7	7	7	7
SD_CLK	Serial Clock	5	5	5	5	5
VDD	3.3V	4	4	4	4	4
VSS	Ground	3, 6	3, 6	6	3, 6	3, 6
Reserved	Reserved	8, 9	8, 9	1, 8	-	-
NC	No Connect	-	10, 11	-	-	-
SD_CI	Card Indication, Optional	Socket Feature	Socket Feature	Socket Feature	Socket Feature	Socket Feature
SD_WP	Write Protect, Optional	Package Feature	-	-	-	-
Operating Voltage	DC Voltage	2.7 – 3.6V	2.7 – 3.6V	2.7 – 3.6V	2.7 – 3.6V	2.7 – 3.6V or 1.65 – 1.95V

Figure 3: Figure Card Pinouts (bottom view)



Functional Description

The SDCard (Secure Digital Memory Card) Component implements a SD/MMC card interface. It uses one digital block in the SPI mode to communicate with an SD Card. It also uses one or more port pins for chip select, card detection, and write protect notification.

The signals between the PSoC and the SD memory card are labeled with respect to the SD Card. There are four required signals that are functionally equivalent to a standard SPI interface and two SD Card specific signals that are optional.

The SD_SCLK signal is the SPI transmit/receive clock. It is one half the clock rate of the input clock signal. The effective transmit/receive bit rate is the input clock divided by two. The input clock is set in the Device Editor Window. During initialization, the SD_SCLK is set to use the 32kHz, then back to the user selected clock afterward. This is per the MMC/SD specification that initialization must be with a clock of less than 400kHz. Due to limitations of the SDCard User Module and the PSoC device itself, the full speed transfer rate of 20MHz for an MMC card or 25MHz for an SD card is not possible.

The SD_DI signal is used to transfer data from the PSoC to the MMC/SD Card. It is equivalent to the SPI Master Out Slave In (MOSI) signal. SD_DO is the signal used to transfer data from the MMC/SD Card to the PSoC. It is equivalent to the SPI Master In Slave Out (MISO) signal. The SD_SCLK clocks the data in both directions and is driven by the PSoC which acts as the master. It is the equivalent to the SPI Serial Clock (SCLK). The fourth signal, SD_CS is used to enable the MMC/SD Card when communicating. The SD_CS signal is asserted low before transmitting a sequence of command and/or data bytes and is returned to the high state after the sequence is completed, ending the transfer. The SD_CD (Card Detect) and SD_WP (Write Protect) are optional signals used to sense if the card is present and if it is write protected.

Allow all API functions to complete before disabling the SDCard Component or turning the power off on the target system. This guarantees that no data is lost or MMC/SD card data is corrupted. Perform a flush and close any open files before disabling the SDCard User Module or removing a card being accessed - this insures that all data transfers are complete.

Configuration

This changes the size of the Component's flash and SRAM usage. The configuration is accessible from the Properties window for the component. This can be accessed by double clicking the SDCard Symbol on the Schematic page or right clicking it and selecting "Properties". There are two settings:

- **MAXFILES** – Sets the maximum number of files that the application is able to open simultaneously. Enough RAM space is reserved to open the specified number of files. When the

Build_Configuration is set to Full File System, each open file requires 24 bytes of RAM. The other four build configurations use 20 bytes per file. Default value is 2.

- **Build_Configuration** – There are six (6) configuration settings. The detailed information and its uses are captured in the table below. Default value is "Full FileSystem"

Table 2: Build_Configuration Options

Build_Configuration Value	Use
Full FileSystem	The Full file system uses the largest amount of Flash and RAM. It also is the most complete file system able to work with both FAT16 and FAT32 formatted flash cards as well as high level commands such as file copy, rename and delete.
Standard FileSystem	The Standard file system is the same as the full version except it does not have FAT32 file system support. Since most cards are formatted as FAT16 this selection saves over 2K of code space while continuing to support most Flash cards.
Basic FileSystem	The Basic file system further reduces the memory requirements by removing FAT32 as well as FileRename and FileCopy. It keeps the FileRemove function because it saves about 500 bytes of flash. Keeping it means that the application is able to create and delete files as needed.
ReadOnly FileSystem	The Read Only file system keeps the FAT16 File System and still reads PC compatible file systems. However, it only reads the flash card files. Use this system in an application for reading configuration files as well as bootloaders.
Basic ReadWrite NoFileSystem	The Basic ReadWrite file system is used when PC compatibility is not required such as On Board flash memory like the iNAND Module. Removing the File System specific API reduces the amount of Flash and RAM required by the SDCard User Module to less than 5K bytes.
Custom Configuration	This option is for advanced users who need to create a custom configuration. This feature is not implemented yet.

Application Programming Interface

The API library functions are the core of the SD/MMC card interface. They are written to use as little RAM and Flash space as possible, while remaining as similar to the standard C functions for file access as memory resources allow. While it is not possible to implement all the standard file IO functions for the SD/MMC interface, a basic subset is included as well as some additional functions to aid in file management.

Table 3: Basic Read/Write Commands

Function	Description
void **SDCard_Start**(void);	Starts the SDCard module.
void **SDCard_Stop**(void);	Stops the SDCard module.
void **SDCard_Select**(uchar Enable);	Selects or deselects the SD Card.

uchar **SDCard_InitCard**(void);	Runs all commands to initialize a card for communications.
uchar **SDCard_fseek**(uchar Fptr, ulong Offset);	Seeks a specific offset into a file.
uchar **SDCard_fgetc**(uchar Fptr);	Returns the next character from the file specified.
uchar **SDCard_fbgetc**(uchar Fptr);	Returns the next buffered character from the file specified. This will produce much faster read times than fgetc when reading only one file at a time.
void **SDCard_clearerr**(uchar Fptr);	Clears the error flags for the file.
uchar **SDCard_ferror**(uchar Fptr);	Returns zero if there is no file error on the specified file, non-zero otherwise.
ulong **SDCard_ftell**(uchar Fptr);	Returns the file offset of the next character to read or write.
uchar**SDCard_ReadSect**(ulong address);	Read a sector.
uchar **SDCard_Present**(void);	Returns `1' if a card is present in the socket, `0' if not.
uchar**SDCard_WriteProtect**(void);	Returns `1' if the card is write protected, `0' if not.
uchar**SDCard_WriteSect**(ulong address);	Writes a sector.
uchar **SDCard_fputc**(uchar Data, uchar Fptr);	Writes a character to a file.
uchar **SDCard_fputs**(char *str, uchar Fptr);	Writes a null terminated string to a file.
uchar **SDCard_fputcs**(const char *str, uchar Fptr);	Writes a null terminated constant string to a file.
uchar **SDCard_fputBuff**(uchar *buff, uint count, uchar Fptr);	Writes count characters from a RAM buffer to a file
uchar**SDCard_fputcBuff**(const uchar *buff, uint count, uchar Fptr);	Writes count characters from a ROM buffer to a file.
void **SDCard_fflush**(uchar Fptr);	Flush the write buffers to a file.

Table 4: Top Level File Functions

Function	**Description**
uchar **SDCard_fclose**(uchar Fptr);	Close the specified file and release the pointer.
uchar **SDCard_fopen**(uchar *Filename, const uchar *Mode);	Opens the supplied file name using the specified mode, and returns the file pointer.
uchar * **SDCard_GetFilename**(uint Entry);	Returns the filename for the specified directory entry.
uint **SDCard_GetFileCount**(void);	Returns the number of files in the root directory.
ulong **SDCard_GetFileSize**(uchar Fptr);	Returns the file size of the specified file.
uchar **SDCard_feof**(uchar Fptr);	Returns non-zero if the specified file is at EOF, `0' otherwise.

Table 5: Top Level Writing Functions

Function	**Description**
uchar **SDCard_Remove**(uchar *Filename);	Delete the named file.
uchar **SDCard_Rename**(uchar *OldFilename, uchar *NewFilename);	Rename the named file.

uchar **SDCard_Copy**(uchar *OldFilename, uchar *NewFilename);	Copy the named file.

File System

These notes are intended as an aid to engineers who want to know more of the low level functionality to develop their applications or designs.

Disk Structure

The structure of an SD/MMC card is the same as a typical hard disk drive and has the following features, assuming it has been DOS/Windows FAT16/32 formatted with a single partition. A standard sector is 512 bytes long.

Partition Table

The first sector of the disk contains the partition table. The table holds information for up to four logical drives. Each logical drive has a 16 byte entry starting at offset 1BE hex. The entry for each logical drive contains the disk size in sectors, the location of the boot sector, and other information. The last two bytes of the sector are the classic 55 AA signature. Most SD/MMC cards only have one logical drive assigned and the SDCard User Module accepts that convention.

Boot Sector

The boot sector contains vital information about the drive it references, such as drive, sector, and cluster size. It also contains the number, size, and type of the File Allocation Table (FAT), and many other pieces of data. Only the first 62 bytes (100 bytes for FAT32) of the boot sector contain data of interest. The remainder is the actual code used to boot the system at power up. The last two bytes of the sector are the classic 55 AA signature.

FAT Tables

The File Allocation Table contains a map of the cluster chains assigned to a file, not sectors. A cluster is a group of sequential sectors that is allocated as one unit. The cluster size is set in the boot sector information. Each FAT entry requires two bytes (little endian) for a FAT16 file system. The FAT always starts with the entry 2. Entries 0 and 1 determine the FAT type. The directory entry for a given file includes the size of the file and its starting entry in the FAT. The starting entry either points to the next cluster in the chain, or FFFF to indicate the end of a chain. Chains are not always sequential. When this

happens, it is called file fragmentation and slows file access. Usually, two copies of the FAT follow the boot sector.

Directory

The root directory typically contains 512 entries of 32 bytes each. When using short filenames (DOS 8.3 format), this means a maximum of 511 files listed, because the first position is reserved for the volume label. (Long filenames do not change the size of the directory, but since they use multiple entries to form one filename, far fewer filenames are allocated.) Each entry is used as a filename, deleted entry, subdirectory, volume label, or a blank entry. The information contained in each entry has filename and extension (or directory name/volume label), file type, file size, starting FAT entry, creation time/date, etc. The filename is eight characters, right padded with spaces, followed by the extension, also right padded with spaces, all in uppercase. Long filenames automatically generate short filenames. If the filename is greater than eight characters, it is truncated to six characters with an added tilde and number to make it unique (e.g. 'FOO.BAR', 'MYFILE~1.TXT', MANUFA~2.XLS'.) The directory follows the FAT tables on the disk.

Note FAT32 directories are treated as files and so the length is variable. The module has a limit of 0xFFFE entries.

File Area

The file area is the remainder of the partition after the end of the directory. It is divided into clusters that correspond to the entries in the FAT table. Please note that to match the FAT table entries, the starting cluster is always cluster two. There are no clusters zero or one.

Note FAT12 uses three nibbles per entry, and therefore, each two entries share a middle byte. This is an older, more complicated, scheme and which is not supported by this user module. FAT32 is similar to FAT16, except that each FAT entry is four bytes long.