SDCard Datasheet SDCard V 1.2

Resources	Build Configurations									
	Full File System		Standard File System		Basic File System		Read Only File System		Basic Read Write	
	Approximate PSoC® Memory Use (bytes)									
	Flash	RAM[a]	Flash	RAM[b]	Flash	RAM[b]	Flash	RAM[b]	Flash	RAM
CY8C29xxx, CY8CLED16	19633	629	17050	613	16437	613	11751	613	4245	575
SDCard Functions Included in the Configuration										
SDCard_clearerr	X		X		X		X		X	
SDCard_fclose	X		X		X		X			
SDCard_feof	X		X		X		X			
SDCard_ferror	X		X		X		X			
SDCard_fflush	X		X		X				X	
SDCard_fgetc	X		X		X		X		X	
SDCard_fbgetc	X		X		X		X		X	
SDCard_fopen	X		X		X		X			
SDCard_fputBuff	X		X		X				X	
SDCard_fputcBuff	X		X		X				X	
SDCard_fputc	X		X		X				X	
SDCard_fputcs	X		X		X				X	
SDCard_fputs	X		X		X				X	
SDCard_fseek	X		X		X		X		X	
SDCard_ftell	X		X		X		X		X	
SDCard_Copy	X		X							
SDCard_GetFileCount	X		X		X		X			
SDCard_GetFilename	X		X		X		X			
SDCard_GetFileSize	X		X		X		X			
SDCard_InitCard	X		X		X		X		X	
SDCard_Present	X		X		X		X		X	

Resources	Build Configurations									
	Full File System		Standard File System		Basic File System		Read Only File System		Basic Read Write	
	Approximate PSoC® Memory Use (bytes)									
	Flash	RAM[a]	Flash	RAM[b]	Flash	RAM[b]	Flash	RAM[b]	Flash	RAM
SDCard_Remove	X		X		X					
SDCard_Rename	X		X							
SDCard_Select	X		X		X		X		X	
SDcard_Start	X		X		X		X		X	
SDard_Stop	X		X		X		X		X	
SDCard_WriteProtect	X		X		X				X	
Comments	Supports FAT16/32		Supports FAT16		Supports FAT16		Supports FAT16		Not PC Compatible	

a. Add 24 bytes of RAM for each file open at the same time (for FAT32 File System)

b. Add 20 bytes of RAM for each file open at the same time (for FAT16 File System)

For one or more fully configured, functional example projects that use this user module go to www.cypress.com/psocexampleprojects.

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 User Module 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 User Module 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 user module 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 User Module functions.

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

1. 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.
2. The SDCard User Module is written in both C and assembly language. You must have a valid C compiler license, even if your target application uses only assembly language.

Figure 1. Interfacing an SD Card to A PSoC Device Operating at 3.3 V

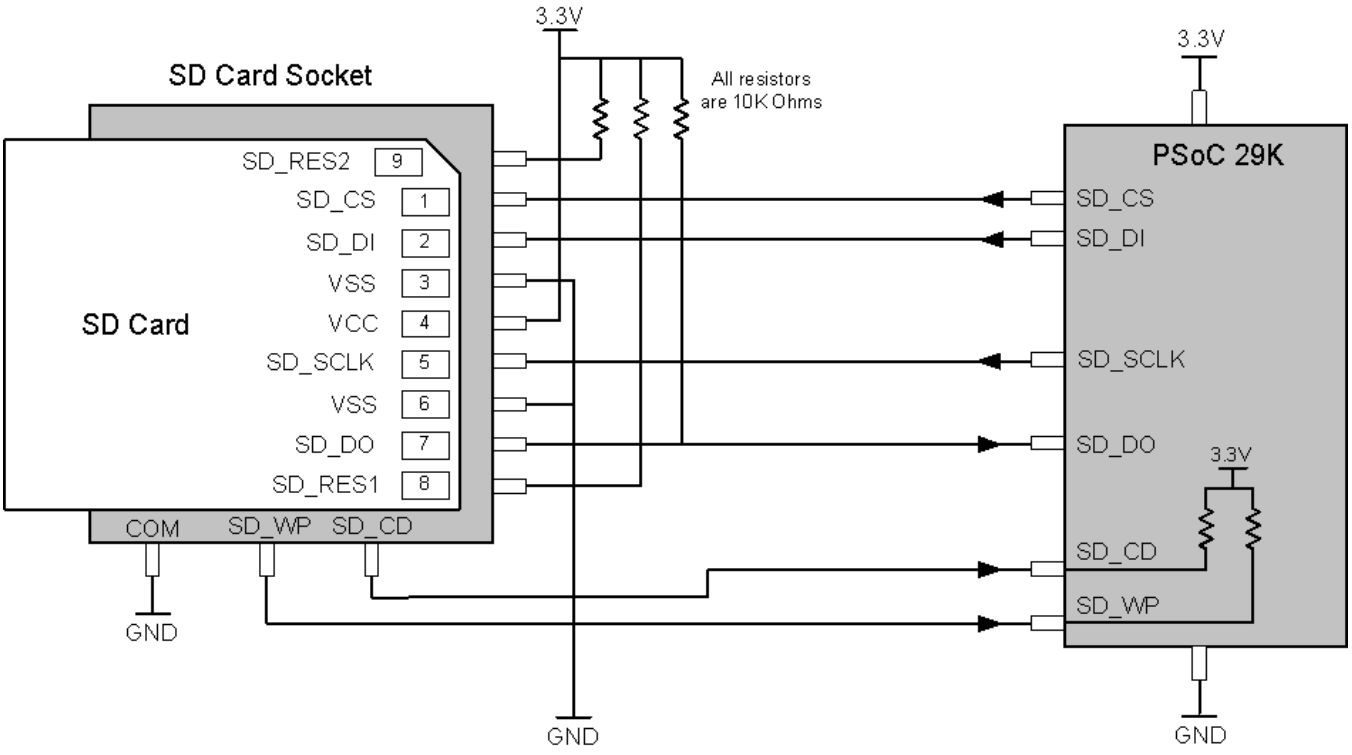


Figure 2. Interfacing an SD Card to A PSoC Device Operating at 5 V

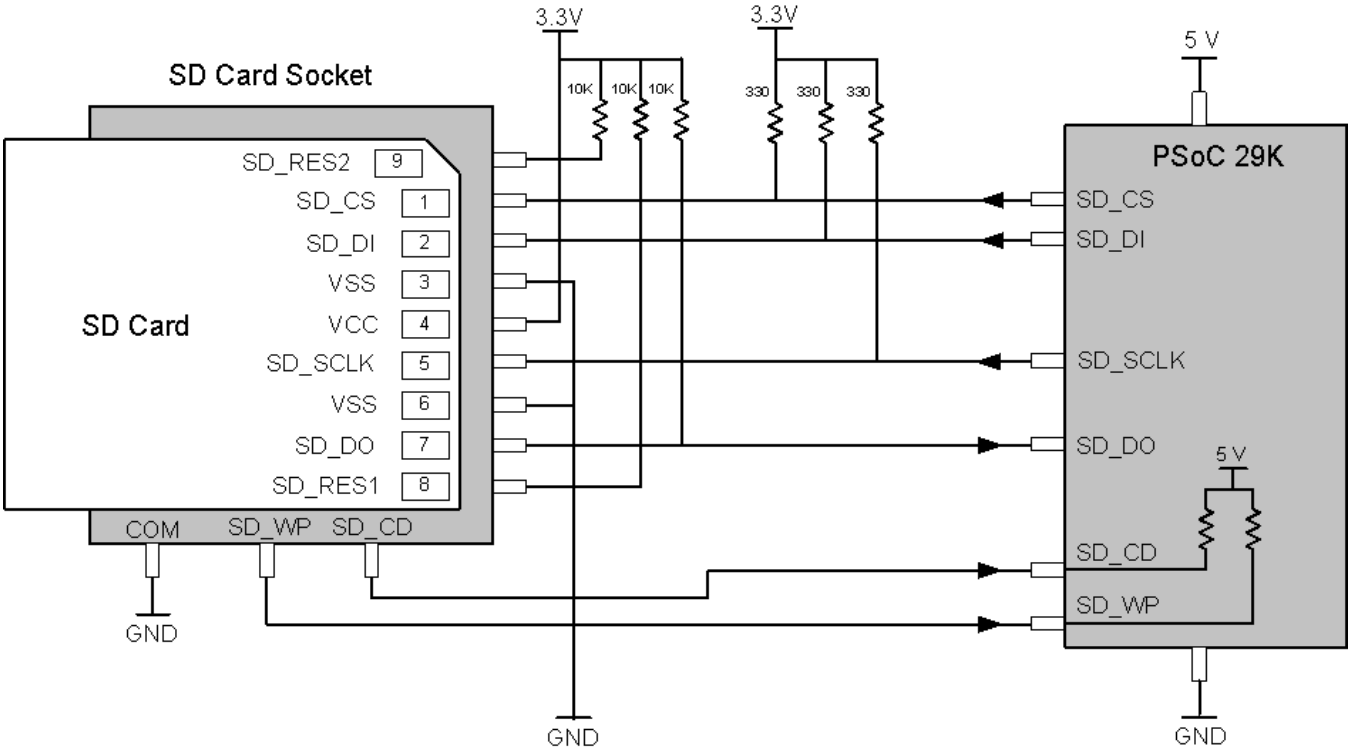


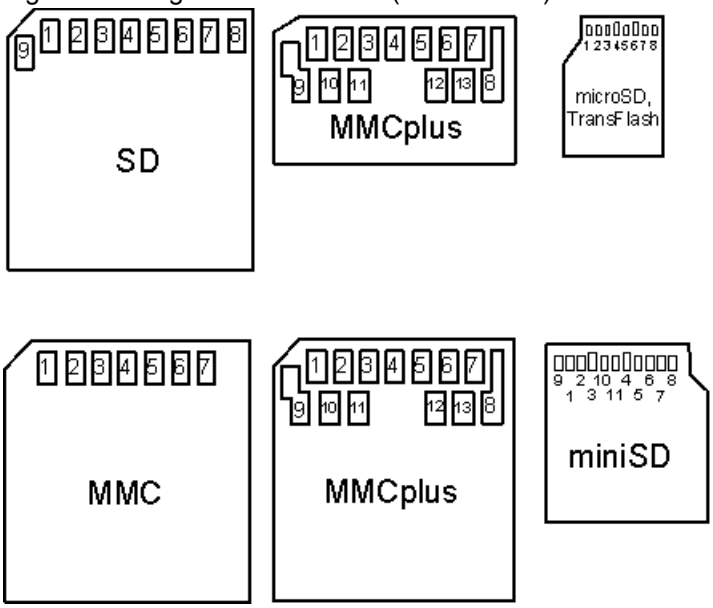
Table 1. PSoC Signals to Flash Card Pins

SD Module Name	Description	SD	miniSD	microSD, TransFlash	MMC	RS_MMS, MMC plus, MMCmobile	PSoC Drive Type	
							3.3V	5V
SD_CS	Card Select	1	1	2	1	1	Strong	*Open Drain low
SD_MOSI	SD_IN PSoC-to-Card Commands and Data	2	2	3	2	2	Strong	*Open Drain low
SD_MISO	SD_OUT	7	7	7	7	7	High Z	High Z
SD_CLK	SD_CLK	5	5	5	5	5	Strong	*Open Drain Low
VDD	3.3V	4	4	4	4	4		
VSS	Ground	3,6	3,6	6	3,6	3,6		
Reserved		8,9	8,9	1,8	-	-		

NC	No Connect	-	10, 11	-	-	-		
SD_CI	Optional						Pull Up	Pull Up
SD_WP	Optional (SD cards only)		-	-	-	-	Pull Up	Pull Up
Supply Voltage	VDC	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		

*330 Ωpull up to 3.3V is required when PSoC is operating at 5 Volts.

Figure 3. Figure Card Pinouts (bottom view)



Functional Description

The SDCard (Secure Digital Memory Card) User Module 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 labelled 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. See the following table:

Signal Name	Description	SPI Equivalent
SD_CS	Card Select (Active Low)	SS
SD_DI	Data Input	MOSI
SD_DO	Data Output	MISO

SD_SCLK	Interface clock	SCLK
SD Card specific signals (optional)		
SD_CD	Card Detect (Active Low)	NA
SD_WP	Card Write Protection (Active Low)	NA

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 User Module 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.

DC and AC Electrical Characteristics

Table 2. SD DC and AC Electrical Characteristics

Parameter	Conditions and Notes	Typical	Limit	Units
SD_SCLK	Maximum bit rate	-	4	MHz
Write Speed	12 MHz CPU Clock	2250		Bytes/Second
Read Speed	12 MHz CPU Clock	2800		Bytes/Second

Set the VCC supply anywhere within the operational voltage range of the SD/MMC cards in use (typically 2.7 - 3.3 VDC). The card and the PSoC microcontroller both operate on the 3.3V VCC. You can run the PSoC at 5V in your design, but it then requires 5V – 3.3V level shifting to interface with the SD/MMC card, adding parts and cost to the design.

Placement

The SDCard User Module maps onto a single PSoC block and may be placed in any of the Digital Communications blocks. Reserve port pins for use by the SD_WP, SD_CD, and SD_CS signals.

Parameters and Resources

Build_Configuration

This changes the size of the flash and RAM used by the SDCard User Module.

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 for the SDCard User Module to meet specific application needs. To use it, modify the supplied **SDCard_Config.h** in the project directory.

Table 3. SDCard_Config.h Symbolic Constants

Symbolic Constant	Use
#define SDCard_MAXFILES xxx	This constant is set automatically by the “Maximum Open Files" paramter in the user module Parameters window and should not be modified in the SDCard_Config.h file.
#define ENABLE_DEBUGFUNCT	Enable for debug functions.
#define ENABLE_FAT32	Add if FAT32 is required. Most flash cards that are formatted as FAT32 can be reformatted as FAT16 (you need the FAT16 and the basic file system).
#define ENABLE_FILESYSTEM	Add if PC compatibility is desired. It enables the FAT16 + Basic file system.
#define ENABLE_FILECOPY	Enable the File Copy function. You must have a file system.
#define ENABLE_FILEREMOVE	Enable File Remove function. You must have a file system.
#define ENABLE_FILERENAME	Enable File Rename function. You must have a file system.
#define ENABLE_WRITE	Used for any writing to the card. Use with or without a file system)

Maximum_Open_Files

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. When Build_Configuration is set to Custom using SDCard_Config.h then both the maximum number of files as well as the amount of RAM needed for each file is set in the SDCard_Config.h file.

Clock

The SDCard User Module is clocked by one of the available sources. Use the global I/O buses to connect the clock input to an external pin or a clock function generated by a different PSoC block. Specify VC1, VC2, VC3, or one of the other clock source options. For optimum performance, set this clock to a clock source of 4 MHz. Set the clock rate to two times the desired bit rate. One data bit is transmitted or received for every two input clocks.

SD_DO

The SD_DO (Data Output) signal should be routed as an input to the PSoC. Once the signal is routed to a pin, set the drive mode for that pin to “Hi Z".

SD_DI

The SD_DI (Data Input) signal should be routed as an output to the PSoC. Once an output pin is selected and the signal is routed, set the drive mode. In a system with both the SD Card power supply and the PSoC supply at 3.3 volts, set the drive mode to “Strong". If the PSoC must be operated at 5 volts and the SD Card at 3.3 volts, set the drive mode to “Open Drain Low" and connect an external pull up of 330 ohms between the SD_DI signal and the 3.3 Volt power supply.

SD_SCLK

The SD_SCLK is the clock for the SD_DI and SD_DO signals. This pin is an output from the PSoC to the SD Card. Once an output pin is selected and the signal is routed, set the drive mode. In a system with both the SD Card power supply and the PSoC supply at 3.3 volts, set the drive mode “Strong". If the PSoC must be operated at 5 volts and the SD Card at 3.3 volts, set the drive mode to “Open Drain Low" and connect an external pull up of 330 ohms between the SD_DI signal and the 3.3 Volt power supply.

SD_CS_Port

Selects the port for the SD_CS signal.

SD_CS_Pin

Selects the pin from the port selected by the SD_CS_Port parameter. The SD_CS (Card Select) is used to enable the SD Card for operation. Route this signal as an output from the PSoC to the SD Card. The drive mode is automatically set as an internal pull up.

SD_CD_Port

Selects the port for the SD_CD signal.

SD_CD_Pin

Selects the pin from the port selected by the SD_CD_Port for the SD_CD signal. The SD_CD (Card Detect) signal senses when a SD Card is inserted into the socket.

SD_WP_Port

Selects the port for the SD_WP signal.

SD_WP_Pin

Selects the pin from the port selected by SD_WP_Port parameter for the SD_WP signal. The SD_WP (Write Protect) pin senses the write protect setting on the SD Card via the SD Card socket.

InvertSD_DO

This parameter should be kept at the “Normal" setting.

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.

Note While the library functions are similar to the standard C functions, they are not in all cases identical. See the entry for each function for more information. Also note that the available commands is dependent on the Build Configuration parameter.

For Large Memory Model devices, it is also the caller's responsibility to preserve any value in the CUR_PP, IDX_PP, MVR_PP, and MVW_PP registers. Even though some of these registers may not be modified now, there is no guarantee that will remain the case in future releases.

Table 4. 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.

Function	Description
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 5. 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 6. 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.

SDCard_clearerr

Description:

Clears the error flags for the file specified by the file pointer.

C Prototype:

```
void SDCard_clearerr(uchar Fptr)
```

Parameters:

Fptr: The file pointer used to access the file.

Return Value:

None.

See Also:

SD_feof, SDCard_ferror

SDCard_Copy

Description:

Copies the source file to the destination file. The source file must exist. The destination file may or may not exist. If the destination file exists but is not empty, its contents are replaced by the contents of the source file.

C Prototype:

```
uchar SDCard_Copy(uchar *OldFilename, uchar *NewFilename)
```

Parameters:

OldFilename: The source file from which to copy.

NewFilename: The destination file to copy to.

Return Value:

Returns zero if successful, nonzero if failure.

See Also:

SD_ferror

SDCard_fbgetc

Description:

Gets the file character currently specified by the file offset and returns it as an unsigned character. It then increments the file offset and adjusts the file control variables to match. The file must be open and have a valid file handle. The difference between fgetc() and fbgetc() is that fbgetc makes use of the sector buffers. When reading a file sequentially it can increase reading speed by more than fgetc().

Note In Basic read/write mode, Fptr is a manually designated index less than MAXFILES assigned by the user to indicate which offset to use.

C Prototype:

```
uchar SDCard_fbgetc(uchar Fptr)
```

Parameters:

Fptr: The file pointer used to access the file.

Return Value:

The current character read from the file.

See Also:

SDCard_fopen, SDCard_fclose, SDCard_fputc, SDCard_fseek, SDCard_ftell, SDCard_GetFileSize

SDCard_fclose

Description:

Closes the file specified by the file pointer. It also releases the file pointer for reuse and clears the file control variables.

C Prototype:

```
uchar SDCard_fclose(uchar Fptr)
```

Parameters:

Fptr: The file pointer used to access the file.

Return Value:

The file error flags as defined in SDCard_ferror.

See Also:

SDCard_fopen, SDCard_ferror

SDCard_feof

Description:

Returns the EOF (End Of File) file error flag for the file specified by the file pointer.

C Prototype:

```
uchar SDCard_feof(uchar Fptr)
```

Parameters:

Fptr: The file pointer used to access the file.

Return Value:

The EOF (End Of File) file error flag for the file specified by the file pointer.

See Also:

SDCard_clearerr, SDCard_ferror

SDCard_ferror

Description:

Returns the file error flags for the file specified by the file pointer.

C Prototype:

```
uchar SDCard_ferror(uchar Fptr)
```

Parameters:

Fptr: The file pointer used to access the file.

Return Value:

One byte containing the file error flags for the file specified by the file pointer. Each open file uses its own file error status byte with the following bit definitions.

Table 7. SDCard_ferror Bits

Bit 7 EOF	Bit 6WE	Bit 5 FPE	Bit FFE	Bit 3 CE	Bit 2 PRE	Bit 1 FNF	Bit 0 IFN
End of file	Write error	File pointer error	File format error	Card error	Parameter range error	File not found	Invalid file name

See Also:

SDCard_clearerr, SDCard_feof

SDCard_fflush

Description:

Takes a file pointer as its argument and empties any internal buffers. The file remains open.

C Prototype:

```
void SDCard_fflush(uchar Fptr)
```

Parameters:

Fptr: The file pointer used to access the file.

Return Value:

None.

See Also:

SDCard_fclose, SDCard_fputc

SDCard_fgetc

Description:

Gets the file character currently specified by the file offset and returns it as an unsigned character. It then increments the file offset and adjusts the file control variables to match. The file must be open and have a valid file handle.

Note In Basic read/write mode, Fptr is a manually designated index less than MAXFILES assigned by the user to indicate which offset to use.

C Prototype:

```
uchar SDCard_fgetc(uchar Fptr)
```

Parameters:

Fptr: The file pointer used to access the file.

Return Value:

The current character read from the file.

See Also:

SDCard_fopen, SDCard_fclose, SDCard_fputc, SDCard_fseek, SDCard_ftell, SDCard_GetFileSize

SDCard_fopen

Description:

Searches the directory for the supplied filename and opens the file using the specified mode. Once a valid file is located, the file control variables are set and the function returns the first unused file pointer for file reference. If the pointer returned equals MAXFILES an error occurred or all available pointers are in use. Use the SDCard_ferror function to check for file errors before accessing the file.

Note If you use the SDCard_GetFilename or other filename routines that use Buffer2 for filename storage, do not use any functions that alter the contents of Buffer2 before calling SD_fopen.

C Prototype:

```
uchar SDCard_fopen(uchar *Filename, const uchar *Mode)
```

Parameters:

Filename: The filename string of the file to open.

Mode: The following modes are permitted:

Mode	Result
r	Open an existing file for input
w	Create a new file, or truncate an existing one, for output.
a	Create a new file, or append to an existing one, for output.
r+	Open an existing file for update (both reading and writing), starting at the beginning of the file.
w+	Create a new file, or truncate an existing one, for update.
a+	Create a new file, or append to an existing one, for update.

Return Value:

The first available file pointer. If the pointer returned is equal to MAXFILES then an error occurred, the filename was not found, or all available pointers are in use.

See Also:

SDCard_fclose, SDCard_GetFileName, SDCard_fputc, SDCard_fgetc, SDCard_fflush, SDCard_fseek, SDCard_ftell

SDCard_fputBuff

Description:

This C function writes n bytes from a RAM buffer to a file.

Note In Basic read/write mode, Fptr is a manually designated index less than MAXFILES assigned by the user to indicate which offset to use.

C Prototype:

```
uchar SDCard_fputBuff(uchar * buff, uint count, uchar Fptr)
```

Parameters:

buff: Pointer to buffer of data in RAM.

count: Count bytes in the buffer to be written to the file.

Fptr: The file pointer used to access the file.

Return Value:

EOF if any error is detected.

See Also:

SDCard_fgetc, SDCard_fflush, SDCard_fseek

SDCard_fputc

Description:

This C function takes a character value and a file pointer as its arguments and writes the character to the file.

Note In Basic read/write mode, Fptr is a manually designated index less than MAXFILES assigned by the user to indicate which offset to use.

C Prototype:

```
uchar SDCard_fputc(uchar Data, uchar Fptr)
```

Parameters:

Data: The character to write to the file.

Fptr: The file pointer used to access the file.

Return Value:

EOF if any error is detected; if not it returns the written character.

See Also:

SDCard_fgetc, SDCard_fflush, SDCard_fseek

SDCard_fputcBuff

Description:

Writes n bytes from a ROM buffer to a file.

Note In Basic read/write mode, Fptr is a manually designated index less than MAXFILES assigned by the user to indicate which offset to use.

C Prototype:

```
uchar SDCard_fputcBuff(const uchar * buff, uint count, uchar Fptr)
```

Parameters:

buff: Pointer to buffer of data in ROM.

count: Count of bytes in the buffer to be written to file.

Fptr: The file pointer used to access the file.

Return Value:

EOF if any error is detected; if not it returns the written character.

See Also:

SDCard_fgetc, SDCard_fflush, SDCard_fseek

SDCard_fputcs

Description:

Writes a constant character string to the file specified by the file pointer parameter.

Note In Basic read/write mode, Fptr is a manually designated index less than MAXFILES assigned by the user to indicate which offset to use.

C Prototype:

```
uchar SDCard_fputcs(const char * Str, uchar Fptr)
```

Parameters:

Str: Pointer to null terminated const character string.

Fptr: The file pointer used to access the file.

Return Value:

EOF if any error is detected.

See Also:

SDCard_fgetc, SDCard_fflush, SDCard_fseek

SDCard_fputs

Description:

Writes a null terminated character string to the file specified by the file pointer parameter.

Note In Basic read/write mode, Fptr is a manually designated index less than MAXFILES assigned by the user to indicate which offset to use.

C Prototype:

```
uchar SDCard_fputs(char * Str, uchar Fptr)
```

Parameters:

Str: Pointer to null terminated character string.

Fptr: The file pointer used to access the file.

Return Value:

EOF if any error is detected.

See Also:

SDCard_fgetc, SDCard_fflush, SDCard_fseek

SDCard_fseek

Description:

Sets the file offset to a user specified value. This allows random access to any area within a file. The file must be open and a valid file pointer supplied.

Note In basic mode, Fptr is the index for which memory space (not file) offset is used.

C Prototype:

```
uchar SDCard_fseek(uchar Fptr, ulong Offset)
```

Parameters:

Fptr: The file pointer used to access the file.

Offset: The number of bytes to offset into the file. Must not exceed the size of the accessed file.

Return Value:

The file error flags as defined in SD_ferror.

See Also:

SDCard_ftell, SDCard_fgetc, SDCard_fputc, SDCard_GetFileSize

SDCard_ftell

Description:

This C function returns the file offset of the next character to read or write. The file must be open and a valid file pointer supplied.

Note In basic mode, Fptr is the index to which memory space (not file) offset to use.

C Prototype:

```
ulong SDCard_ftell(uchar Fptr)
```

Parameters:

Fptr: The file pointer used to access the file.

Return Value:

An unsigned long containing the read position offset.

See Also:

SDCard_fseek, SDCard_fgetc, SDCard_fputc

SDCard_GetFileCount

Description:

Returns the number of valid files in the root directory.

C Prototype:

```
uint SDCard_GetFileCount(void)
```

Parameters:

None.

Return Value:

The number of valid files in the root directory.

See Also:

SDCard_GetFilename

SDCard_GetFilename

Description:

Returns a pointer to a filename from the directory using a passed index. For instance, an index of five returns the fifth valid filename in the directory. Call SD_GetFileCount first to make certain that you do not index beyond the valid choices.

Note The pointer returned by SDCard_GetFilename() is a pointer to a string in scratchpad memory . Read and save this filename string immediately if it is needed later on in the program. Otherwise, Buffer may get overwritten and the filename deleted.

C Prototype:

```
uchar *SDCard_GetFilename(uint Entry)
```

Parameters:

Entry: The entry number of the directory entry. A value of five opens the fifth valid file in the root directory. Files are not sorted by name.

Return Value:

A character pointer to the filename string (11 characters plus null).

See Also:

SDCard_GetFileCount, SDCard_fopen

SDCard_GetFileSize

Description:

Returns the total number of bytes in the file specified. Call it to make sure that the file size is not zero or that any user functions are able to handle a file of this size.

C Prototype:

```
ulong SDCard_GetFileSize(uchar Fptr)
```

Parameters:

Fptr: The file pointer used to access the file.

Return Value:

An unsigned long containing the number of bytes in the entire file.

See Also:

SDCard_fopen, SDCard_fgetc, SDCard_ftell

SDCard_InitCard

Description:

Does a low level communication initialization of the SD card, sets the card data mode to SPI, detects the card type and file system installed, and updates the global variables accordingly. Call this function anytime a card is inserted or a serious card error is detected. This function returns a byte that has information on card type and format type.

Note FatType needs to have a value of 0x20 (FAT16 formatting) before using the card.

C Prototype:

```
uchar SDCard_InitCard(void)
```

Parameters:

None.

Return Value:

A byte containing:

Card (lower nibble) 0=None detected, 1=MMC, 2=SD.

FAT (upper nibble) 00=None, 0x10=FAT12, 0x40=FAT16, 0xB0=FAT32.

See Also:

SD_Present

SDCard_Present

Description:

Reports whether or not a card is inserted into the SD card socket. It does not however indicate that the card is working properly. Use it to detect whether or not the card is inserted or removed, whether or not to call the SD_InitCard function, or generate errors.

Note This function is optional as not all SD sockets support a card insertion signal.

C Prototype:

```
uchar SDCard_Present(void)
```

Parameters:

None.

Return Value:

One for card present, zero for card absent.

See Also:

SDCard_InitCard

SDCard_ReadSect

Description:

Reads one absolute sector. The address passed must be the beginning of a valid sector boundary. Valid sector boundaries are zero and multiples of 512.

C Prototype:

```
uchar SDCard_ReadSect(ulong address)
```

Parameters:

address: The sector address to read.

Return Value:

Returns zero if the sector read was successful, nonzero otherwise.

SDCard_Remove

Description:

Deletes the named file. The file must exist and not be open.

C Prototype:

```
uchar SDCard_Remove(uchar *Filename)
```

Parameters:

Filename: The name of the file to delete.

Return Value:

Zero if successful, nonzero if failure.

See Also:

SD_fclose, SD_ferror

SDCard_Rename

Description:

Changes the name of OldFilename to NewFilename. The file must exist and not be open.

C Prototype:

```
uchar SDCard_Rename(uchar *OldFilename, uchar *NewFilename)
```

Parameters:

OldFilename: The current name of the file.

NewFilename: The new name of the file.

Return Value:

Zero if successful, nonzero if failure.

See Also:

SD_ferror

SDCard_Select

Description:

This function is used to enable or disable the communication interface between the PSoC and the SD Card.

C Prototype:

```
void SDCard_Select(uchar Enable)
```

Parameters:

Enable: Selects if card is enabled or disabled (1 = Enable, 0 = Disable)

Return Value:

None.

See Also:

SDCard_Stop, SDCard_Start

SDCard_Start

Description:

Starts the user module by initializing the library code. Run SDCard_Start before using any other SDCard library functions or SD hardware accessed.

C Prototype:

```
void SDCard_Start(void)
```

Parameters:

None.

Return Value:

None.

See Also:

SDCard_Stop

SDCard_Stop

Description:

Stops the PSoC hardware setup. Run SDCard_Stop should after all SD library functions are finished and after stopping all SD card access.

C Prototype:

```
void SDCard_Stop(void)
```

Parameters:

None.

Return Value:

None.

See Also:

SDCard_Start

SDCard_WriteProtect

Description:

Reports whether or not an SD card's write protect tab is enabled to prevent writes to the SD card.

Note This function is optional and is only supported on standard SD cards. If this function is desired be sure to use an SD socket that supports write protect detection.

C Prototype:

```
uchar SDCard_WriteProtect(void)
```

Parameters:

None.

Return Value:

One if write protected, zero if write enabled.

See Also:

SD_Present

SDCard_WriteSect

Description:

Writes one absolute sector. The address passed must be the beginning of a valid sector boundary. Valid sector boundaries are zero and multiples of 512.

C Prototype:

```
uchar SDCard_WriteSect(ulong address)
```

Parameters:

address: The sector address to write.

Return Value:

Returns zero if the sector write was successful, nonzero otherwise.

Sample Firmware Source Code

```
//-------------------------------------------------------------------------
// Hello World Sample Project
//
// Description:
// This is a simple example of what it take to write a string to a SD Card File.
//
// Program Flow:
// 1) Wait for card to be inserted.
// 2) Initializes card interface.
// 3) Opens file "hello.txt" for writing.  If not there it creates the file.
// 4) Writes the string "Hello World" with a CR and LF.
// 5) Closes the file. Then goes back to step '1' and waits for card to
//    be inserted.
//
// Connections:
//
// In this example the SDCard UM digital block is placed
// at DCB02.  The signals are connected to the follow pins.
// The drive mode settings assume that both the SD Card and
// the PSoC are operating at 3.3 Volts.
//
// Signal     Port   Select    Drive     Note
// SD_DO   => P2[3]  GlobalIn  High Z    Data Out
// SD_SCLK => P2[2]  GlobalOut Strong    Clock In
// SD_DI   => P2[1]  GlobalOut Strong    Data In
// SD_CS   => P2[0]  StdCPU    Strong    Card Select (Active Low)
// SD_CD   => P2[4]  StdCPU    Pull Up   Card Detect (Active Low)
// SD_WP   => P2[5]  StdCPU    Pull Up   Write Protect (Active High)
//
// A LED User Module was used to drive the busy LED on the
// CY3210-SDCARD demo board at port P2[6].  The Drive is set to
// "Active High" and the instance name is "BusyLED".
//-------------------------------------------------------------------------
```

```
#include <m8c.h>        // part specific constants and macros
#include "PSoCAPI.h"    // PSoC API definitions for all user modules

#define  CARD_PRESENT      1
#define  CARD_NOT_PRESENT  0

char helloFile[] = "hello.txt";

void main(void)
{
   char   cardInfo;                    // Card information
   char   fp;                          // File Pointer
   BYTE   CardState, oldCardState;     // Card inserted state variables

   BusyLED_Off();                       // Make sure LED is Off
   oldCardState = CARD_NOT_PRESENT;     // Initialize card insertion state
   SDCard_Start(  );                   // initialize hardware and SDCard_lib buffers

   while(1)
   {
      CardState = SDCard_Present();     // Get current card insertion state
      if(CardState != oldCardState)     // Check for a change
      {
         oldCardState = CardState;      // Save last state
         if(CardState == CARD_PRESENT)  // Card inserted
         {
            BusyLED_On();               // About to talk to card, turn on busy LED
            SDCard_Select(SDCard_ENABLE); // Select card
            cardInfo = 0;
            while ( ! cardInfo )        // Wait for card to communicate
            {
               // initialize card, determine card type and file system type
               cardInfo = SDCard_InitCard();
            }

            fp = SDCard_fopen(helloFile,"a");      // Open file to append data
            SDCard_fputcs("Hello World\r\n", fp);  // Write string
            SDCard_fclose(fp);                     // Close file
            SDCard_Select(SDCard_DISABLE);         // Deselect card
            BusyLED_Off();                         // Turn off busy LED
         }
      }
   }
}
```

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.

Version History

Version	Originator	Description
1.2	DHA	Added Version History

Note PSoC Designer 5.1 introduces a Version History in all user module datasheets. This section documents high level descriptions of the differences between the current and previous user module versions.