



Access SD Memory Cards (Part 1)

Solid-State Storage Media in Embedded Apps

Jeff uses SD solid-state storage media in embedded designs. In this series of articles, he provides a thorough introduction to SD technology. Read on to learn how he gets real data in and out of the SD card with the FAT file format.

My first hard drive was 5 MB and took up only a 5" bay. Considering double-sided 5" floppies were 720 KB at the time, that was pretty darn good. Solid-state memory is beginning to give mechanical storage a run for its money. The memory card in my camera holds 1,000 times more data than that first hard drive. We're measuring in gigs not megs today with terabytes available from some hard drive manufacturers.

By far the most popular device these days is the solid-state USB memory device. Used for back-ups and file exchanges, it's named after the part of your anatomy it most closely represents, the thumb. However, the wafer-thin SD, miniSD, and microSD are some of the physically smallest devices available (see Figure 1). The microSD is about as small as I'd want a memory card, or else it would surely be lost among my pocket lint. When you're designing a portable device, small is what you're aiming for. So, it makes sense to consider SD memory when specifications call for external memory.

An SD memory card uses 512-byte blocks with a potential 32-bit (double word) address space, so you can dump a lot of data to and from one of these little guys. If all of the reading and writing is handled within your device, you can take charge of how and where data is stored. However, if you want to remove the SD card and read the data with another device, you better adhere to the rules of one of the

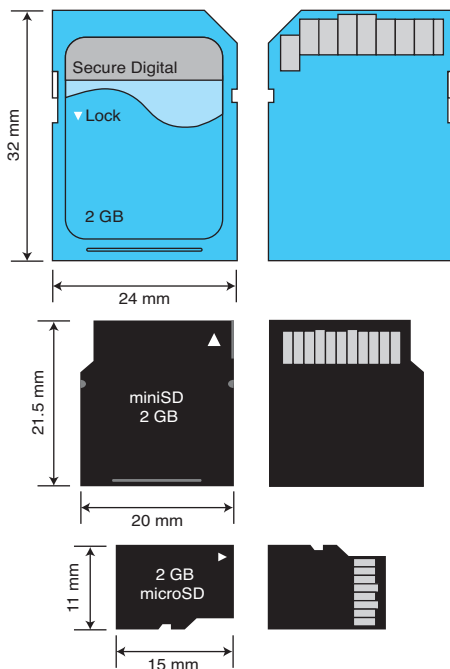


Figure 1—This shows the relative card sizes available for SD cards. Card adapters allow smaller sized cards to be read in standard SD card sockets. However, many universal card readers will accept all sizes without an adapter. (Source: http://en.wikipedia.org/wiki/Secure_Digital_card)

Pins	SD Mode			SPI Mode		
	Name	I/O Type ¹	Description	Name	I/O Type	Description
1	CD/DAT3	I/O /PP	Card detect/data line[Bit3]	CS	I	Chip select (negative true)
2	CMD	PP	Command/response	DI	I	Data in
3	V _{SS1}	S	Ground	V _{SS}	S	Ground
4	V _{DD}	S	Supply voltage	V _{DD}	S	Supply voltage
5	CLK	I	Clock	SCLK	I	Clock
6	V _{SS2}	S	Ground	V _{SS2}	S	Ground
7	DAT0	I/O /PP	Data line[Bit0]	DO	O/PP	Data out
8	DAT1	I/O /PP	Data line[Bit1]	RSV	—	Reserved*
9	DAT2	I/O /PP	Data line[Bit2]	RSV	—	Reserved*

[1] S stands for power supply, I stands for input, O means output, I/O means bidirectionally, and "PP" stands for I/O using push-pull drivers.

[*] These signals should be pulled up by the host side with 10- to 100-k Ω resistance in SPI mode.

Table 1—The SD card interface has a multiple-bit-wide data path. The particulars for using this interface are available to paying members of the SD Card Association. However, the alternative SPI has been released as a "simplified specification" to the general public. (Source: Toshiba, www.toshiba.com/taec/components/Datasheet/020725_SD-Mxxx.pdf)

most widely used file format systems. The FAT file system began with the efficient FAT12 model, which was used on floppy disks and was limited to about 32 MB of addressable space, and was more than anyone was thinking about at the time. The larger FAT16 model was used on hard drives, and could handle up to about 2 GB. In the late 1980s, as hard drive sizes began approaching 2 GB, the FAT32 model was created, which included 32-bit sector counts, expanding the range to 2 TB. Although FAT32 has brought on slightly larger capacities, it also exhibited out some of its inefficiencies via pokey behavior with large sizes. At that point, a rash of incompatible file systems entered the picture, leaving the FAT as the reigning champ of compatibility.

SD INTERFACE

My newest laptop and desktop both have built-in card readers for a number of different solid-state memory card formats. Although the maximum capacities of these devices continues to increase, the amount of space you need for your application probably doesn't come anywhere near these capacities. So, in reality, having gigs of space is most likely overkill for your projects.

Staying compatible with the FAT file system makes a lot of sense. If you have been reading my columns each month, you will remember when I presented a small two-line LCD serial terminal (RS-232), which got me out of a design jam of no longer having

serial ports on any of my computers ("Serial Terminal Solution," *Circuit Cellar* 219, 2008). At the time, I said I

would be using the same schematic again in a future project. As you may remember, it had a card reader interface that I ignored for that project. It's time to put it to good use.

An SD memory card uses a proprietary SPI-type interface. I say SPI-type interface because communication uses multiple bidirectional data lines, instead of a single data line, and a SPI clock. Commands require a single I/O-bit command/response structure. However, there are 4 more bits of data I/O (see [Table 1](#)). The SD card interface also supports the standard three-wire SPI. (Needless to say, this will have a slower throughput.) Without special hardware interfacing to support the native (multi-I/O) mode, you will need to spend a lot of valuable time bit banging individual bits of your code,

Address	00	02	04	06	08	0A	0C	0E	ASCII	
0910	0000	0000	0000	0000	0000	0000	0000	00FE	!...!
0920	0000	0000	0000	0000	0000	0000	0000	0000
0930	0000	0000	0000	0000	0000	0000	0000	0000
0940	0000	0000	0000	0000	0000	0000	0000	0000
0950	0000	0000	0000	0000	0000	0000	0000	0000
0960	0000	0000	0000	0000	0000	0000	0000	0000
0970	0000	0000	0000	0000	0000	0000	0000	0000
0980	0000	0000	0000	0000	0000	0000	0000	0000
0990	0000	0000	0000	0000	0000	0000	0000	0000
09A0	0000	0000	0000	0000	0000	0000	0000	0000
09B0	0000	0000	0000	0000	0000	0000	0000	0000
09C0	0000	0000	0000	0000	0000	0000	0000	0000
09D0	0000	0000	0000	0000	0000	0000	0000	0000
09E0	0000	0000	0000	0000	0000	0000	0000	0000
09F0	0000	0000	0000	0000	0000	0000	0000	0000
0A00	0000	0000	0000	0000	0000	0000	0000	0000
0A10	0000	0000	0000	0000	0000	0000	0000	0000
0A20	0000	0000	0000	0000	0000	0000	0000	0000
0A30	0000	0000	0000	0000	0000	0000	0000	0000
0A40	0000	0000	0000	0000	0000	0000	0000	0000
0A50	0000	0000	0000	0000	0000	0000	0000	0000
0A60	0000	0000	0000	0000	0000	0000	0000	0000
0A70	0000	0000	0000	0000	0000	0000	0000	0000
0A80	0000	0000	0000	0000	0000	0000	0000	0000
0A90	0000	0000	0000	0000	0000	0000	0000	0000
0AA0	0000	0000	0000	0000	0000	0000	0000	0000
0AB0	0000	0000	0000	0000	0000	0000	0000	0000
0AC0	0000	0000	0000	0000	0000	0000	0000	0000
0AD0	0000	0000	0000	0000	0000	0000	0000	0802
0AE0	0600	C101	85C2	0000	7B00	1D97	0000	0000	{.....
0AF0	0000	0000	0000	0000	0000	0000	0000	0000
0B00	0000	0000	0000	0000	0000	0000	0000	0000
0B10	0000	0000	0000	0000	0000	0000	5500	AFAAU..
0B20	0079	0000	0000	0000	0000	0000	0000	0000	y.....

Figure 2—This is a dump of a data input buffer after a read of "Block 0" of the SD card. After an FE block count byte, 512 bytes of data are highlighted. Note the first partition entries highlighted at offset 0x1BE. The block ends with the marker 0x55 and 0xAA. A 16-bit CRC follows the marker.

so using the three-wire SPI is a great alternative. Standard SPI hardware will reduce the servicing of the interface to once a byte, as opposed to managing every bit.

Although all of the connections necessary for bus mode are supported in this schematic, I will be using the three-wire SPI mode. You can find SD card connectors available for all three sizes of the SD physical format. I used the standard SD connector here. When I purchased a microSD card, it came with an adapter that accepts microSD cards and enables them to be plugged into a standard size connector.

If you remember, the microcontroller used in this design operates on 3.3 V. This gives you a straightforward interface to SD cards so you don't have to worry about logic-level translation. The multi-I/O bus interface requires a clock output and five bidirectional I/Os. The CMD I/O is used to receive commands and send responses. In addition, four DAT lines are available on the interface, including DAT (0:3) for bidirectionally moving data. Alternate support for the three-wire SPI uses CMD to receive commands, data (SDO), DAT0 for sending responses, data (SDI), and the clock line (SCLK).

SD INITIALIZATION

The three-wire mode is not the native interface bus mode after a reset, so you must first establish a request for three-wire SPI mode by holding the DAT3 line low while issuing a CMD0. If the SD card recognizes this event, it will stop bus mode operation and respond on DAT0 (instead of the CMD). Any further communication will use the three-wire SPI mode (until power loss). Although the default state of CRC checking in the SPI mode is disabled, the initial CMD0 command must have the proper CRC because the default state in bus mode is CRC enabled.

Commands have a fixed length of 6 bytes. Response lengths are determined

MBR offset	Parameter size	Value found	Description	Meaning
0x01BE	Byte	0x00	0x00 = Inactive 0x80 = Active	Volume boot record has no boot code
0x01BF	3 Bytes	0x000802	Beginning of partition CHS	Cylinder/head/sector address = 0x000802
0x01C2	Byte	0x06	0x00 = Unknown 0x01 = FAT12 0x04 = FAT16 0x05 = Ext MSDOS 0x06 = FAT16 0x0B = FAT32 0x0C = see 0x0B 0x0E = see 0x06 0x0F = see 0x05	Using a 16-bit FAT
0x01C3	3 Bytes	0xC2C101	End of partition CHS	Cylinder/head/sector address = 0xC2C101
0x01C6	Double word	0x00000085	Sector count from MBR to partition	Partition starts at sector 0x85
0x01CA	Double word	0x001D977B	Sector count of partition	Partition is 0x001D977B sectors in length

Table 2—The first logical sector on the card holds the partition information. This tells you how the SD card is formatted, where to look for (this partition's) volume boot sector (VBS), and the partition's size.

by the command issued. Most responses are a single byte; however, 2- and 5-byte responses are also common, depending on the amount of information being passed. All responses begin with a cleared most significant bit (MSB). This distinguishes itself from a nonresponse or delayed response. Because an intelligent controller controls the SD card internally, there will be potential delays as it exercises the necessary actions required for each command. Therefore, data clocked in from the SD card may contain one or more bytes of data where the MSB is set. You must continue to clock in data until a byte is read with the MSB cleared. Timeouts are therefore defined by a maximum number of extra bytes read without receiving a legal byte. Also, it must be noted that because data is clocked out and in at the same time, at least one extra output byte must be sent to enable an input response to be returned (clocked in).

Initialization of the SD card after power has been applied is required prior to any data transfers. Besides sending CMD0 (while DAT2 is low, used as *CS) to initialize the interface for three-wire SPI, additional commands are required to determine the type of SD card that has been inserted. Commands CMD8, ACMD41, and CMD58 help determine minimum operating voltages and whether the card has finished its initialization. Refer to the SD Card Association's 2006 document titled "SD Specifications Part 1: Physical Layer Simplified Specification" for a complete list of commands.

Once initialized, you may request additional information about the card by using CMD9 to retrieve the card-specific data (CSD) and CMD10 to retrieve the card identification (CID). The CID includes manufacturer and product information. The CSD includes maximum speed and current requirements, as well as block length (size in bytes), capacity (number of blocks), file format to expect, and other parameters. This information is enough for you to access the card using your own format for data (i.e., just dumping continuous data to contiguous blocks). To use the FAT file format, I will investigate how it is structured by looking at an SD card that has been preformatted.

I'm a "show me" kind of guy. Reading all about how something works (or is supposed to work) is fine for a light overview. But to really know something, I must work with it. That's why I like using an in-circuit debugger. I can stop program execution at any point and see what's going on. I will use this feature to investigate the FAT16 file system on my microSD card. Now that the SD card has successfully been initialized, you need to add only one additional command to see some data. CMD17 is a read single-block command. It returns up to the maximum block length (512) bytes of data after a special response byte of "FE" used as a block marker and before a 2-byte (16-bit) CRC (for a maximum total of 515 bytes). Note that CMD18, the multiple block read, automatically decrements this block

Address	00	02	04	06	08	0A	0C	0E	ASCII	
0910	0000	0000	0000	0000	0000	0000	0000	EBFE!
0920	9000	2020	2020	2020	2020	0200	0120	0200
0930	0200	0000	EDF8	3F00	2000	8500	0000	7B00?{
0940	1D97	8000	2900	CEB4	2F7D	4F4E	4E20	4D41)	}NO NAM
0950	2045	2020	4620	5441	3631	2020	0020	0000	E FAT 16	...
0960	0000	0000	0000	0000	0000	0000	0000	0000
0970	0000	0000	0000	0000	0000	0000	0000	0000
0980	0000	0000	0000	0000	0000	0000	0000	0000
0990	0000	0000	0000	0000	0000	0000	0000	0000
09A0	0000	0000	0000	0000	0000	0000	0000	0000
09B0	0000	0000	0000	0000	0000	0000	0000	0000
09C0	0000	0000	0000	0000	0000	0000	0000	0000
09D0	0000	0000	0000	0000	0000	0000	0000	0000
09E0	0000	0000	0000	0000	0000	0000	0000	0000
09F0	0000	0000	0000	0000	0000	0000	0000	0000
0A00	0000	0000	0000	0000	0000	0000	0000	0000
0A10	0000	0000	0000	0000	0000	0000	0000	0000
0A20	0000	0000	0000	0000	0000	0000	0000	0000
0A30	0000	0000	0000	0000	0000	0000	0000	0000
0A40	0000	0000	0000	0000	0000	0000	0000	0000
0A50	0000	0000	0000	0000	0000	0000	0000	0000
0A60	0000	0000	0000	0000	0000	0000	0000	0000
0A70	0000	0000	0000	0000	0000	0000	0000	0000
0A80	0000	0000	0000	0000	0000	0000	0000	0000
0A90	0000	0000	0000	0000	0000	0000	0000	0000
0AA0	0000	0000	0000	0000	0000	0000	0000	0000
0AB0	0000	0000	0000	0000	0000	0000	0000	0000
0AC0	0000	0000	0000	0000	0000	0000	0000	0000
0AD0	0000	0000	0000	0000	0000	0000	0000	0000
0AE0	0000	0000	0000	0000	0000	0000	0000	0000
0AF0	0000	0000	0000	0000	0000	0000	0000	0000
0B00	0000	0000	0000	0000	0000	0000	0000	0000
0B10	0000	0000	0000	0000	0000	0000	5500	78AAU,x
0B20	00D8	0000	0000	0000	0000	0000	0000	0000

Figure 3—This is a dump of a data input buffer after a read of Block 0x85 of the SD card. After an FE block count byte, 512 bytes of data are highlighted. Note the VBS entries highlighted at offset 0. The block ends with the markers 0x55 and 0xAA. A 16-bit CRC follows the marker.

marker, sending another 515 bytes, until a CMD12 (stop transmission) is sent. However, I'll stick with the single-block read for now.

DESIGNING WITH FAT

One advantage of using an established format for storing files is that all of the decision making has been done for you. You can put more of your effort into the application instead of how and where the data will be stored in memory. If you follow the basic rules, your data will be available to every computer/device that supports the file format. You can find in-depth documents describing the FAT file format on the 'Net, so I will not go into every detail here. I will touch

only on those areas that apply directly to this project and those that I believe are of interest to others attempting to implement FAT in their own projects.

There are five areas of interest here: the master boot record (MBR), the boot sector, the FAT region, the root directory region, and the data region. The MBR is the name given to the first sector of the drive. It may contain boot code (boot from disk) for the computer and has four possible partition entries located at fixed locations within the record. All commands consist of 6 bytes: the command byte, a 4-byte argument, and a CRC byte. To read this sector from the SD card, a CMD17 is issued for LBA sector 0 with a command argument, in this

case, an address of 0x00000000.

The actual data that was transferred into my 515-byte buffer shows no boot code after the FE block marker byte at 0x91E (see Figure 2). The first partition entry is at offset 0x1BE (0x91F+0x1BE) or location 0xADD. Table 2 shows the meaning of the values found in this partition entry beginning at buffer location 0xADD. This is easy to decipher because you are dealing with the logical block address (LBA). You need only to know that this device is formatted using FAT16 with the partition beginning at LBA location 0x85 and spanning 0x1D977B sectors.

The second area of interest is the volume boot sector (VBS) (see Table 2). I didn't explain how to set up the argument of CMD17 earlier when I wanted LBA sector 0 (the MBR) because sector 0 is at address 0. But LBA sector 0x85 is not address 0x85! It's actually 0x200 (the sector size) times the LBA sector number. If you remember, a sector consists of 512 or 0x200 bytes, so LBA sector 0x85 is really byte address 0x200 (sector size in bytes) times 0x85 (LBA of the sector required) or 0x00010A00. The command argument is a double word address value of the first byte of interest.

The dump of LBA 0x85 shows the first 3 bytes (boot jump vector) as a near jump to 0000 or no boot code, right after the FE block marker byte at 0x091E (see Figure 3). I am interested in the following 64 or so bytes. Refer to Table 3 to glean the good stuff out of this sector.

The third area of interest is the FAT region. Referring to Table 3, a FAT contains 237 sectors and there are two of them. This means the FAT region takes up a total of 0x1DA sectors (237 × 2). The first FAT begins after the VBS (one reserved sector). That's at LBA 0x86 (the second at 0x173 (0x86 + 0xED)). The next area of interest is the root directory region. Because the FAT requires 0x1DA sectors, the root directory will begin at 0x260:

$$[0x86(\text{first sector of the FAT}) + 0x1DA(\text{sector length of the FATs})] \quad [1]$$

Each sector in the root directory

contains 16 directory entries:

$$\left[\frac{512 \text{ bytes (sector size)}}{32 \text{ bytes (directory length)}} \right] \quad [2]$$

The root directory uses 32 sectors:

$$\left[\frac{512 \text{ (Entries)}}{16 \text{ (Entries/sector)}} \right] \quad [3]$$

This enables you to calculate the last area of interest, the data region. The data region follows the root directory. It begins at LBA 0x280:

$$\left[0x260 \text{ (first sector of the Root Directory)} + 0x20 \text{ (Sector length of the Root Directory)} \right] \quad [4]$$

You don't really need to know this, but I think it helps to know the boundaries.

You now have a complete map of each area in the FAT file system (see Table 4). With a bit more background information on the FAT and directory structures, you'll have enough information to find a directory or file and see where it's located in the SD card.

USER INTERFACE

I want to stop discussing the FAT file system here because the number of figures I need is rapidly exceeding the number allotted. Instead, I will finish this article with a discussion about how I use the three push buttons and a 2 × 20 LCD to access the SD card. With a limit of three push buttons, each of these might require some awkward list of functions that depend on the mode or screen being displayed. This would not be user friendly and could end up as a labeling nightmare.

In sticking with a format I used previously, the LCD will be used to display a single member of a list of items. The list might be the names of all the files and subdirectories in the present directory or a list of functions to choose from. The first line displays the item and the second line labels the button functions. This usually takes the form of "1-Next 2-Prev 3-OK."

A mechanical switch on the SD card socket indicates when a device has been inserted by removing the ground from the input and allowing card detect (CD) to rise (thanks to a pull-up resistor). When the circuit is powered on and the microcontroller initializes, a call is made to the `display main menu` routine before returning to the keyscan loop. (Note that this is the basic process of this application: display something and then give the user a chance to respond.) The `display main menu` will give you a chance to scroll through the main menu list and

choose either the property mode (to alter the parameters of the serial port, data rate, number of data bits, parity, number of stop bits, and flow control) or the operational mode (to use the SD card interface). The `keyscan` (or `main`) loop is used to monitor key presses and jump to the associated function routine, one for each button. The `main` loop is also responsible for checking the CD input for a card insertion.

If CD indicates that there is no card in the socket, the initialized flag is cleared and the "display operational menu" is called before entering the keyscan loop. If CD indicates that a card is in the socket, a check is made to determine if the card has already been initialized. If the inserted card has already gone through initialization, then execution jumps directly to the keyscan loop. (This is the normal program flow.) If the SD card has not been initialized, then a card initialization routine and the `display operational menu` routine is called before entering the keyscan loop.

STATE MACHINE

Note that all roads (calls) lead (return) to the keyscan loop, where any key press steers execution temporarily to a call where other actions take place based on what mode (state) the program is in. I've previously described mode 0 or the "display main menu." In this mode, a press of button 1 calls function 1 (mode 0). This will increment the operational/properties mode choice, and the "display main menu" is called before returning to the keyscan loop. If button 2 is pressed, function 2 (mode 0) will decrement the operational/properties mode choice and the `display main menu` routine is called before returning to the keyscan loop. Pressing button 3 calls function 3 (mode 0). If the choice was properties, then the mode would be changed to mode 7 and the `display properties menu` routine is called before returning to the keyscan loop. If the choice was operational,

VBS Offset	Parameter size	Value found	Description	Meaning
0x00	3 Bytes	0xEB0090	Jump instruction	No boot code
0x03	8 Bytes	" "	OEM Name	Blank
0x0B	Word	0x0200	Bytes/sector	Sector = 512 bytes
0x0D	Byte	0x20	Sectors/cluster	Cluster = 32 sectors
0x00	Word	0x0001	Reserved sector count	One sector
0x10	Byte	0x02	FAT Count	Two FAT copies
0x11	Word	0x0200	Maximum root directory entries	512 Root directory entries
0x13	Word	0x0000	Total sector count	0 (Use offset 0x20)
0x15	Byte	0xF8	Media descriptor	Hard disk
0x16	Word	0x00ED	FAT Sector count	FAT Length = 237 sectors
0x18	Word	0x003F	Sectors/track	Track = 64 sectors
0x1A	Word	0x0020	Head count	Heads = 32
0x1C	Double word	0x00000085	Hidden sectors	VBS = 0x85
0x20	Double word	0x001D977B	Sector count	Sectors = 1939323
0x24	Byte	0x80	Physical drive	Drive = 128
0x25	Byte	0x00	Reserved	
0x26	Byte	0x29	Extended boot signature	
0x27	Double word	0x2F7DCEB4	Serial number	796774068
0x2B	11 Bytes	"NO NAME"	Volume label	
0x36	8 Bytes	"FAT16"	FAT Type	

Table 3—LBA 0x85 holds information for the VBS. You can calculate the beginning LBA for each FAT and the root directory.

then the mode is changed to mode 8 and the display operational menu is called before returning to the keyscan loop.

I'll skip over the following list of modes because they were covered in Issue 219 and deal only with setting the properties of the serial port ("Serial Terminal Solution," *Circuit Cellar*, 2008). Mode 7 (display properties menu) consists of a list of five choices: data rate, data bits/parity, stop bits, flow control, and exit. Mode 1 (display data rate menu) has eight choices: 300, 1,200, 2,400, 9,600, 19,200, 38,400, 250,000, and exit. Mode 2 (display bits menu) has five choices: 8N, 9N, 8O, 8E, and exit. Mode 3 (display stop menu) consists of a list of three picks: 1, 2, and exit. Mode 4 (display flow menu) enables you to choose from three items: None, Hardware, and exit.

New to this project are Mode 5 (display file menu), Mode 6 (display directory menu), and Mode 8 (display operational menu). I'll wrap up this column by

LBA	Length	Area
0x00	One sector	Master boot record
0x85	One sector	Volume boot sector
0x86-0x172	0xED Sectors	First FAT region
0x173-0x25F	0xED Sectors	Second FAT region
0x260-0x27F	0x20 Sectors	Root directory region
0x280-0x1D9800	0x1D9580 Sectors	Data region

Table 4—The MBR and VBS give you the information required to create a complete LBA sector map of the entire FAT file system.

discussing Mode 6, which was mentioned earlier when I discussed the CD input and SD card detection.

Except for the initialization process of the SD card (when the media is accessed to determine if it can be supported), should there be a problem Mode 6 handles much of the SD card access and displays a directory from the card or an error message. As you saw earlier, the data on the SD card is held in 512-byte sectors or blocks. Reading the first sector (MBR) clues you into which block holds the VBS. In the VBS, you find out about the makeup of the SD card and where the FAT and Root directory regions lie.

Once the Root directory is read, you can interpret the directory entries. Any directory or file name you come across is added to the directory buffer. This buffer holds a list of all of the names found in the root directory region (files or subdirectories).

Completion of the directory search leads to a call to the display buffer routine. This routine displays the first subdirectory or file name in the buffer and returns to the keyscan loop (enabling you to scroll up and down through the buffer's list of entries and eventually choose one). This is the only time that the list does not wrap around. I thought that wrapping around this list could get confusing, especially if the list is long, so all directory lists scroll but do not wrap around.

Should any of the SPI communication commands have problems reading or interpreting the data from the SD card, a "bad media" flag is raised leading to an LCD message suggesting that the

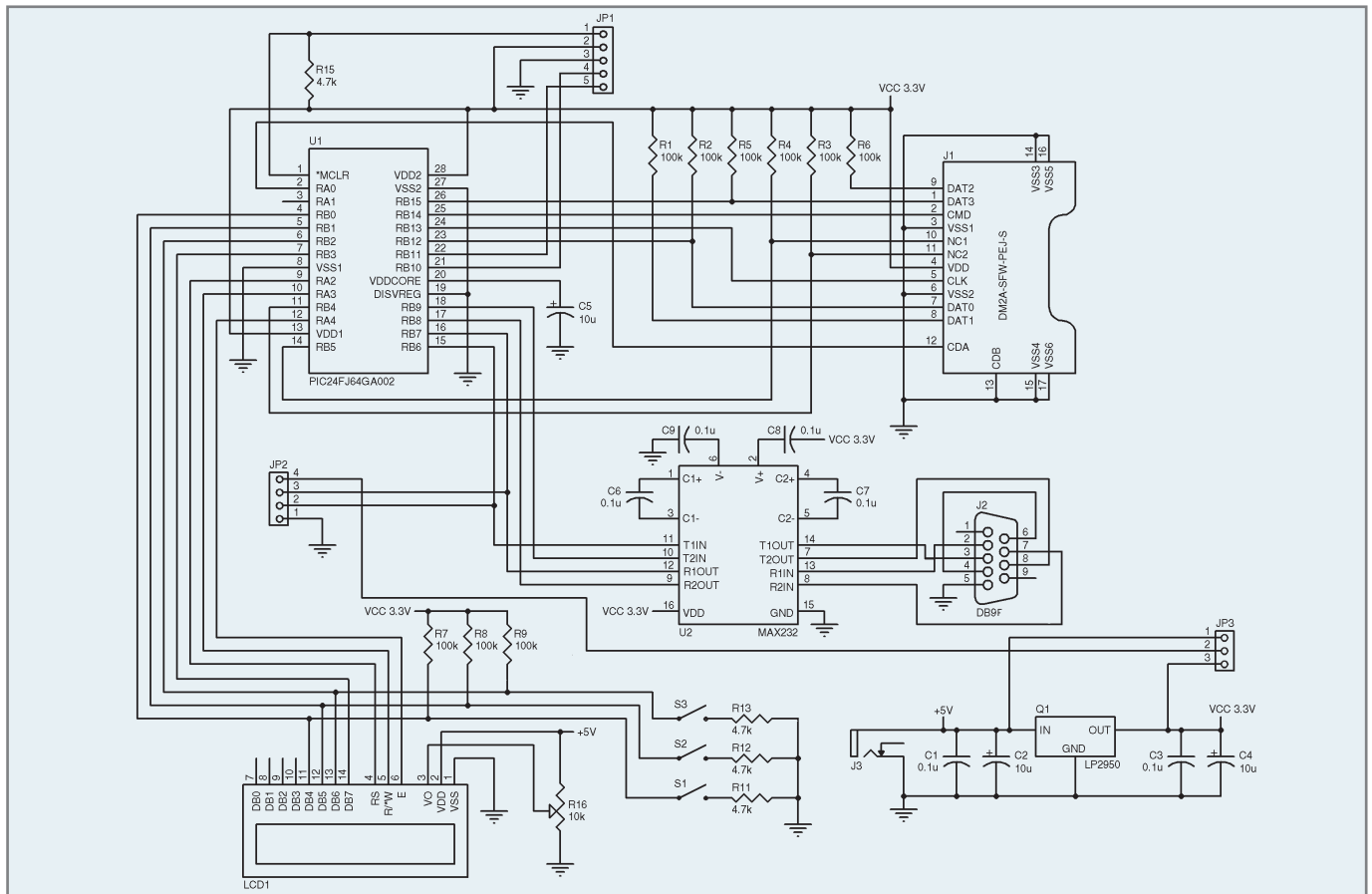


Figure 4—The Microchip Technology PIC24FJ64GA002 microcontroller used here is operated on 3.3 V, eliminating any unnecessary logic-level translation with the SD card interface. The 2 x 20 character LCD runs on 5 V, but the microcontroller can handle that interface without a hitch, thanks to 5-V-level-tolerant I/O.

media be replaced. Should the media be removed at any time, the "media ready" flag is cleared, leading to an LCD message asking for media to be inserted.

BREAK

You're now on the brink of choosing a file or subdirectory from the SD card. There is far more to discuss than I have room for here. You need to investigate the directory structure and see how to determine what's there, learn what the FAT is used for, and figure out how to locate a file's data. There is so much more to come, yet you've come so far already. You learned about the physical interface used in accessing an SD card, commands that it recognizes, and how to use these commands to investigate the media.

Refer to [Figure 4](#) if you want to begin playing around with this circuit. Next month, I will put the code for this project on the *Circuit Cellar* FTP site when I finish examining the actions needed to get real data out of and into the SD card using the FAT file format. ☒

Jeff Bachiochi (pronounced BAH-key-AH-key) has been writing for Circuit Cellar since 1988. His background includes product design and manufacturing. You can reach him at jeff.bachiochi@imaginethatnow.com or www.imaginethatnow.com.

RESOURCES

Microsoft Corp., "Microsoft Extensible Firmware Initiative: FAT32 File System Specification; FAT: General Overview of On-Disk Format," Version 1.03, 2000, www.microsoft.com/whdc/system/platform/firmware/fatgen.mspx.

SanDisk Corp., "SanDisk Secure Digital Card Product Manual," Version 1.9, 80-13-00169, 2003, www.cs.ucr.edu/~amitra/sdcard/ProdManualSDCardv1.9.pdf.

SD Card Association, "SD Specifications Part 1: Physical Layer Simplified Specification," Version 2.00, 2006, www.sdcard.org/developers/tech/sdcard/pls/Simplified_Physical_Layer_Spec.pdf.

Toshiba America, Inc., "SD - Mxxx Series SD Memory Card," 2002, www.toshiba.com/taec/components/Datasheet/020725_SD-Mxxx.pdf.





SOURCE

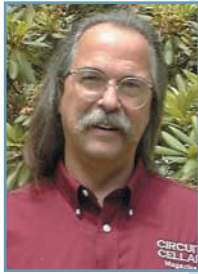
PIC24FJ64GA002 Microcontroller

Microchip Technology, Inc. | www.microchip.com

APEC 2009 THE PREMIER
February 13–21, 2009 GLOBAL EVENT
Washington, DC IN POWER
ELECTRONICS™
Visit the Apec 2009
web site for the latest
information!
www.apec-conf.org

SPONSORED BY



Access SD Memory Cards (Part 2)

Use the FAT File Format to Move Data

Jeff concludes his introduction to SD technology with information about using the FAT file format to move data. He also describes how to create directories and file entries, and delete, save, and append data using ring buffers.

Whether it is audio, video, graphic, database, system, executable, or any number of other file types, they can all be stored, moved, and copied from one storage medium to another, thanks to the operating system's ability to understand how each medium handles the data. While we can't necessarily execute an OS-specific file across multi-OS platforms, many platforms support multiple file systems. So, we can at least exchange data via several supported types of media.

When floppies took over from tapes as the storage medium of choice, DOS introduced us to the FAT file system. At the time, clever designers used shortcuts (space-saving data packing) to cram the most data onto the available media. This inherently put maximum physical limits on what the system could handle. With file sizes measured in kilobytes, this limit was of no concern at the time. Today's needs show how this may have been a bit shortsighted. Improvements to the original FAT file system have met those needs for now. Can we expect this to remain adequate? It services its intended need well, and while we should never say never, I believe it will continue to be supported for many

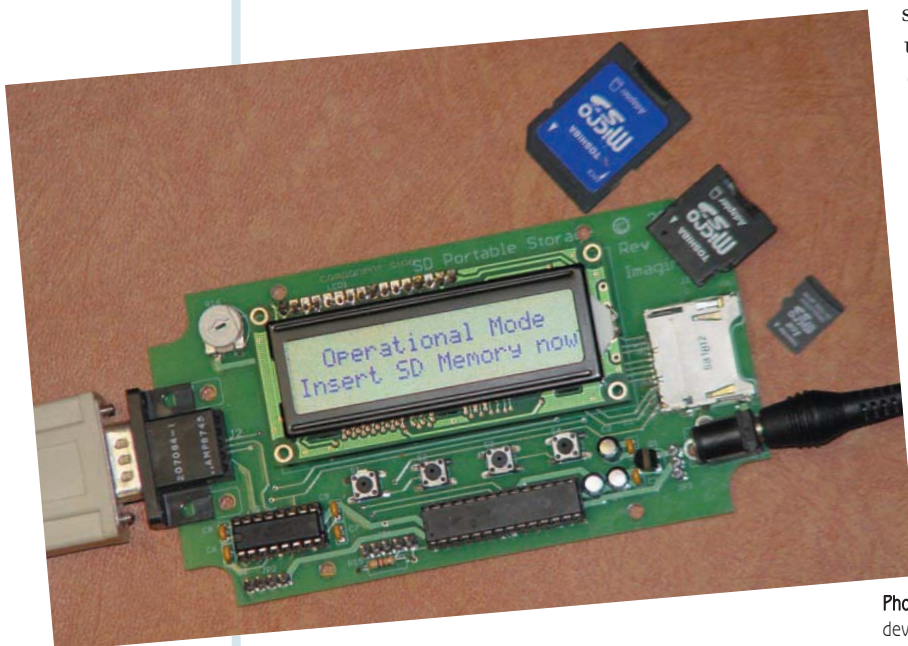


Photo 1—Like everything else, even removable solid-state memory devices have undergone a shrinking process. SD cards are great for adding file storage to your project.

Address	HEX								ASCII	
0910	1309	00FF	FFFF	FF00	0000	0000	210B	FE53!..S
0920	4420	4D49	4352	4F20	2020	0800	0000	0000	D MICRO
0930	0000	0000	00BC	5D0B	3900	0000	0000	0046].	9.....F
0940	494C	4531	2020	2054	5854	2000	0000	0000	ILE1 T	XT
0950	001C	3900	0040	5819	3902	0004	0000	0046	..9..@X.	9.....F
0960	494C	4532	2020	2054	5854	2000	0000	0000	ILE2 T	XT
0970	0000	0000	0000	0000	0000	0000	0000	004FO
0980	4E45	2020	2020	2020	2020	1000	0000	0000	NE
0990	0000	0000	0000	0000	0004	0000	0000	004AJ
09A0	4546	4620	2020	204A	5047	2018	3D50	5819	EFF J	PG =PX.
09B0	3919	3900	0025	4F59	3803	00EB	2700	0042	9.9..%OY	8...!..B
09C0	5300	7500	6E00	7300	6500	0F00	0274	002E	S.u.n.s.	e...t.
09D0	006A	0070	0067	0000	0000	00FF	FFFF	FF01	.j.p.g.
09E0	4800	6100	6C00	6600	2000	0F00	0244	006F	H.a.l.f.	...D.o
09F0	006D	0065	0020	0061	0000	0074	0020	0048	.m.e. a	...t..H
0A00	414C	4644	4F7E	314A	5047	2000	9371	5819	ALFDO~1J	PG ..qX.
0A10	3919	3900	00D9	5946	360C	0015	5928	00E5	9.9...YF	6...Y(..
0A20	3938	3136	3838	5F20	2020	1000	AF1B	181A	981688_
0A30	391A	3900	001C	181A	39AE	0000	4000	0000	9.9....	9...@...
0A40	0000	0000	0000	0000	0000	0000	0000	0000
0A50	0000	0000	0000	0000	0000	0000	0000	0000
0A60	0000	0000	0000	0000	0000	0000	0000	0000
0A70	0000	0000	0000	0000	0000	0000	0000	0000
0A80	0000	0000	0000	0000	0000	0000	0000	0000
0A90	0000	0000	0000	0000	0000	0000	0000	0000
0AA0	0000	0000	0000	0000	0000	0000	0000	0000
0AB0	0000	0000	0000	0000	0000	0000	0000	0000
0AC0	0000	0000	0000	0000	0000	0000	0000	0000
0AD0	0000	0000	0000	0000	0000	0000	0000	0000
0AE0	0000	0000	0000	0000	0000	0000	0000	0000
0AF0	0000	0000	0000	0000	0000	0000	0000	0000
0B00	0000	0000	0000	0000	0000	0000	0000	0000
0B10	0000	0000	0000	0000	0000	0000	0000	008D

Figure 1—This 512-byte sector dump of the first sector of the root directory shows a few 32-byte directory entries. The first entry (beginning at address 0x091F) is reserved for the optional volume name, in this case "SD MICRO." The remaining entries consist of two text files, a subdirectory, and two digital image files. Note that these data dumps are depictions of the memory, as displayed by the ICD2 debugger, address data is in.

years because it is so ingrained into today's media.

Last month, I started discussing how you can use SD solid-state storage media in your embedded designs. Although the intended nibble interface is open to those who join the SD Association, a simplified version of the physical layer specification is available at www.sdcard.com. This interface is a standard SPI. While the throughput might be slower (1 bit versus 4 bits), it works well with a microcontroller's standard SPI hardware. Although I consider the USB thumb drive to be the most popular file transfer media, most cameras, phones, PDAs, MP3 players, and other portable electronics use a smaller form of solid-state storage. The SD card is widely used and available. This makes it a perfect match for embedded products (see [Photo 1](#)).

If you haven't read the first article in this series, please take a few minutes to review the points I covered last month ("Access SD Memory Cards (Part 1): Solid-State Storage Media in Embedded Apps," *Circuit Cellar* 222, 2009). The FAT file system is divided into four sections: the reserved region, the FAT region, the root directory region, and the file and directory data region. The reserved section contains the boot sector or

BIOS parameter block with definitions of various media parameters including where to find the other regions. The FAT region is a list of tags indicating the status of each cluster or group of sectors. The root directory region contains a list of files or directory names and corresponding information. The file and (sub) directory region hold the actual file data or additional subdirectories.

With the interfacing and protocols introduced last month, I had room to describe only the first of four sections of the FAT file system. This month, I will begin by digging into the root directory region using the FAT16 format. From the reserved region, I previously determined a number of important parameters. For your reference, there are 512 entries in a directory. There are 237 sectors in each of two FATs. The first FAT begins in sector 0x86. There are 512 bytes per sector. There are 32 sectors per cluster, and there are 32 bytes per directory entry. Because I know that the FAT begins at sector 0x86 and each of the two FATs are 237 (0xED) sectors each, the root directory must be at 0x86 + 0xED + 0xED or sector 0x260. I'll start by looking at this sector or the logical block address (LBA).

ROOT DIRECTORY, FAT REGIONS

A dump of LBA 260 (the first root directory sector in [Figure 1](#)) shows nine entries: a volume ID, four files, two directories (one deleted), and two entries associated with a long file name. All of the directory entries, whether they are file or directory entries, take on the same format. When they are initially formatted, all directory entries contain 0x00s. A 0x00 as the first byte of directory entries indicates that the entry is empty and there are no more used entries in the rest of the directory. (This fact saves search time.) A file name (or directory name) must consist (for the most part) of one or more alphanumeric characters. Although long file names can exceed eight characters, I will not discuss them here. (You can read about how they are handled in most overviews of the FAT file system.) The directory structure begins with an eight-character name and a three-character extension. A file name has an implied "." between its name and extension. This is determined by the twelfth byte in the directory entry. The attribute byte contains flag bits indicating, among other things, if the entry is a file or a directory. Other bytes contain various time and date information. Besides the name and attribute parameters, the last two are significant. The cluster value tells you where to look for the first sector of the file or subdirectory and the FAT location associated with the entry. The file length indicates the file size in bytes. (Directories always have a zero file length as can empty files.)

The first sector (of either copy) of a FAT region holds 256

16-bit pointers for the cluster numbers 0x00 to 0xFF (with the following sectors holding additional pointers for the remaining clusters). The 16-bit value stored at each cluster position begins life as zero. (Except for cluster 0 and cluster 1, these are reserved and can not be used.) A value of zero means the cluster is not in use. When a directory entry is created, its cluster low word value (at offset 0x1a in the directory entry) points to the first cluster used by the entry. In Figure 1, cluster 2 was assigned to directory entry FILE.TXT, a text file of 4 bytes. This refers to the cluster where the data of the file begins and the FAT location where more information is held. When the entry is a directory or a file that is less than 16,384 bytes, it does not require more than a single cluster (32 sectors) (see Table 1). Therefore, the 16-bit FAT entry for cluster 0x0002 contains a value that indicates that this is the last cluster in the file with a value of (0xFFFF-0xFFFF). Otherwise, the 16-bit FAT entry for cluster 0x0002 would contain the value of the next cluster used by the file. Unless a cluster is the last, each FAT entry would then point to an additional cluster, and so on.

In the dump of the first sector of the FAT, you can see how this works (see Figure 2). The sector data begins after the block byte at address 0x90E. The first few words are 0xFFFFs and 0xFFFFs. The third word is for cluster 0x0002. The 0xFFFF indicates that this is the last cluster. I added a large file (HALF-DO~1.JPG) so you can see how the chaining works. In the root directory, this file was assigned cluster 0x000C (12th). The twelfth word in the FAT entry is not 0xFFFF, but 0x000D. This file's data does not end in cluster 0x000C but continues on into cluster 0x000D. Now look at the thirteenth FAT word and you will see it doesn't end there but each one points to another cluster up to cluster 0x00AD. This FAT entry is 0xFFFF, indicating that this is the last cluster of the file. Back in the last 4 bytes of this file's entry in the root directory sector, you can see that the file's size for this file is 0x00285915 (2,644,245 bytes) and requires many clusters.

Additional clusters were allocated sequentially for this file, but they don't

Address	HEX									ASCII	
0910	1309	00FF	FFFF	FF00	0000	0000	210B	FEF8	
0920	FFFF	FFFF	FFFF	FFF8	FFF8	FFF8	FFF8	FFF8	
0930	FFF8	FFF8	FFF8	FF0D	000E	000F	0010	0011	
0940	0012	0013	0014	0015	0016	0017	0018	0019	
0950	001A	001B	001C	001D	001E	001F	0020	0021	
0960	0022	0023	0024	0025	0026	0027	0028	0029	."#\$.%	.&'(.)	
0970	002A	002B	002C	002D	002E	002F	0030	0031	.*+,-	.../0.1	
0980	0032	0033	0034	0035	0036	0037	0038	0039	.234.5	.678.9	
0990	003A	003B	003C	003D	003E	003F	0040	0041	...;<=	.>?.@A	
09A0	0042	0043	0044	0045	0046	0047	0048	0049	.B.C.D.E	.F.G.H.I	
09B0	004A	004B	004C	004D	004E	004F	0050	0051	.J.K.L.M	.N.O.P.Q	
09C0	0052	0053	0054	0055	0056	0057	0058	0059	.R.S.T.U	.V.W.X.Y	
09D0	005A	005B	005C	005D	005E	005F	0060	0061	.Z[\^]	^_`a	
09E0	0062	0063	0064	0065	0066	0067	0068	0069	.b.c.d.e	.f.g.h.i	
09F0	006A	006B	006C	006D	006E	006F	0070	0071	.j.k.l.m	.n.o.p.q	
0A00	0072	0073	0074	0075	0076	0077	0078	0079	.r.s.t.u	.v.w.x.y	
0A10	007A	007B	007C	007D	007E	007F	0080	0081	.z{. }	~.....	
0A20	0082	0083	0084	0085	0086	0087	0088	0089	
0A30	008A	008B	008C	008D	008E	008F	0090	0091	
0A40	0092	0093	0094	0095	0096	0097	0098	0099	
0A50	009A	009B	009C	009D	009E	009F	00A0	00A1	
0A60	00A2	00A3	00A4	00A5	00A6	00A7	00A8	00A9	
0A70	00AA	00AB	00AC	00AD	00FF	FF00	0000	0000	
0A80	0000	0000	0000	0000	0000	0000	0000	0000	
0A90	0000	0000	0000	0000	0000	0000	0000	0000	
0AA0	0000	0000	0000	0000	0000	0000	0000	0000	
0AB0	0000	0000	0000	0000	0000	0000	0000	0000	
0AC0	0000	0000	0000	0000	0000	0000	0000	0000	
0AD0	0000	0000	0000	0000	0000	0000	0000	0000	
0AE0	0000	0000	0000	0000	0000	0000	0000	0000	
0AF0	0000	0000	0000	0000	0000	0000	0000	0000	
0B00	0000	0000	0000	0000	0000	0000	0000	0000	
0B10	0000	0000	0000	0000	0000	0000	0000	000E	

Figure 2—This is the first sector of the first FAT. This sector holds all of the 16-bit (for FAT16) FAT entries for cluster 0x00–0xFF (successive sectors hold successive cluster entries). Note that the cluster entry in a directory entry (for a subdirectory or file) points to the cluster where the entries actual data resides and also to an entry in the FAT. Any FAT entry with a value greater than 0xFFFF signifies this is the last cluster used by the subdirectory or file. Other values greater than 1 (0 and 1 are reserved) point to the next cluster used by the subdirectory or file.

have to be. The chaining process can jump to any unused cluster. This is a good time to mention what happens when something is deleted. If a single byte—the first character of a directory entry—is changed to 0xE5, the file (or subdirectory) is considered unavailable. Note that all of the information including the actual data has not been altered in any way. This makes every deleted file potentially recoverable, unless it is damaged by data that has been subsequently written to the disk. Thus, remember that deleting a file does not delete the information. Apply a full format to totally wipe the media, not just a quick erase.

PROJECT OBJECTIVES

I could have wimped out and just

added the necessary functions to do logging to and dumping from a file already on the SD card. But, I wanted to make this project as helpful as possible. I added functions that help show how things are done but certainly aren't necessary for this project. The 2 × 20 LCD and three-button interface really made this a challenge. Assuming your formatted SD card has no files on it, you would need to be able to create an entry in the root directory region. This project gives you three choices: create a file, create a (sub) directory, or exit without doing anything. I use the top line of the LCD to give you a choice and the second line to indicate the function of the buttons. Usually, this is button 1, display the next item, button 2, display the previous item, and button 3, choose the item. Let's create

Directory entry offset	Parameter size	Value found	Description	Meaning
0x00	11 Bytes	"FILE1 TXT"	Name	"FILE1 .TXT"
0x0B	Byte	0x20	Attributes Bit 0 = Read only Bit1 = Hidden Bit2 = System Bit3 = Volume ID Bit4 = Directory Bit5 = Archive Bits0:3 = Long name	Archived file
0x0C	Byte	0x00	Reserved	
0x0D	Byte	0x00	Creation time (tenths)	
0x0E	Word	0x0000	Creation time Bits0:4 = Seconds/2 Bits5:10 = Minutes Bits11:15 = Hours	
0x10	Word	0x0000	Creation date Bits0:4 = Day Bits5:8 = Month Bits9:15 = Year + 1980	
0x12	Word	0x391C	Date of last access	8/28/2008
0x14	Word	0x0000	Reserved	
0x16	Word	0x5840	Time of last write	11:02:00 AM
0x18	Word	0x3919	Date of last write	8/25/2008
0x1A	Word	0x0002	Cluster low word	Cluster = 2
0x1C	Double word	0x00000004	File size (bytes)	File size = 4

Table 1—Each directory entry can hold useful information about a file or directory.

a (sub) directory in the root directory region (see Figure 3).

Entering a directory name is the first challenge. I can take advantage of a previously required routine to move 20 characters from the RAM into row 1 of the LCD. The blinking cursor never shows up on the LCD because the LCD print statement leaves it beyond the last visible character position. By moving the cursor to character position 1–20, it can be used as a prompt to you. Buttons 1 and 2 cycle forward and backward through legal characters (including a blank space and backspace arrow for correcting mistakes). Button 3 advances the cursor to the next position (or backspaces). The entry routine exits after eight characters are entered.

All of this entry is worthless if we don't have any open clusters, so we'd better check the FAT region. I use three basic routines with the FAT: locate a FAT entry (cluster), read the cluster entry, and write the cluster entry. I want to find an unused FAT entry (read 0x0000) and claim it for the new directory (write 0xFFFF). The offset into the FAT where the unused entry is found becomes the cluster number for the new directory. Note that directories require a single cluster and therefore will not require chaining in the FAT. A second copy of the FAT is kept for safety measures. While this isn't normally used, it's a good idea to keep it updated for compatibility.

Now the new cluster becomes the home for the new second-level directory. You must add two directory entries to this clean slate, the "dot" and "dot dot" directory entries. The "." directory uses the new cluster number as its cluster pointer. The ".." directory uses the parent cluster as its cluster pointer.

One last operation is necessary before this new directory

can be used. You must go back to the parent directory and add the directory name (and pertinent data) to an empty directory entry. This completes the chain that can point us to the new directory's cluster.

NEW COMMANDS

Adding a file to a directory is actually easier than adding a directory. But there are a few differences. The file name includes a three-character extension. This is indicated on the LCD by a "." separating the eight-character name from the extension. The FAT is handled the same way. (At this point, the file size is zero.) Because the file size is zero, nothing needs to be placed anywhere in the new cluster. An empty directory entry needs to be filled in with the file name and pertinent data.

Assuming a new directory and file have been created (exist) in the root directory cluster, a number of new commands become available. The appropriate commands are offered when a directory or a file name is chosen. When a directory is selected, you can change to the new directory, delete the directory, add a new directory, add a new file, or return to the present directory. When a file is selected, you can delete the file, add a file, dump a file, log to a file, add a new directory, or return to the present directory.

Execution spends most of its time in the operational mode. This is where all the file and directory names in the present directory are displayed (one at a time). When an SD card is inserted, this will always be the root directory (in this case, that's LBA 0x0260). If a directory name is selected (e.g., ONE) and you choose to switch directories, the entry's first cluster pointer (0x0004) is used as the cluster to be used as the present directory. (Refer to Figure 1 and you will see that directory ONE uses cluster 0x0004.) Figure 4 shows a dump of this

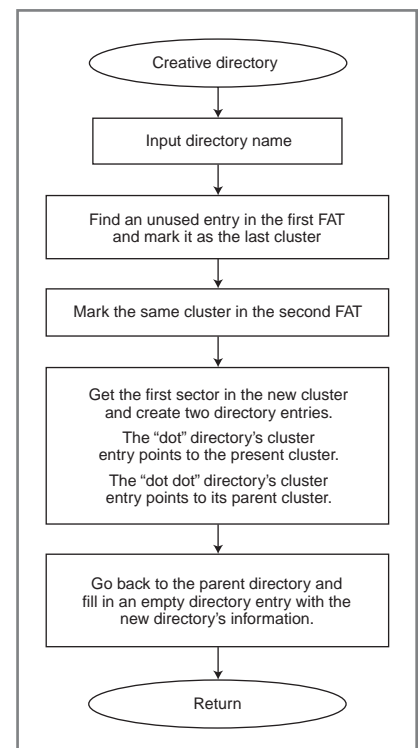


Figure 3—This flowchart greatly simplifies the process for creating a directory. When searching the FAT, for instance, you may need to search every word in every sector of every cluster (up to the maximum 237 sectors per FAT) for an entry just to find out that there is no room left. Obviously, the search loops are exercised more as the media fills up.

directory's first sector, LBA = 0x0003 (Cluster-1) × 0x20 (sectors per cluster) + 0x0260 (root directory) or 0x02C0. You can see the "dot," "dot dot," and (sub) directory TWO entries in the ONE (sub) directory. Note the first cluster word locations (at offset 0x1A of each directory entry pointing to the respective clusters of those entities). By changing the first character 0x54 of directory TWO to 0x0E5, this entry would be eliminated. When using this application to delete a directory, no check is made as to which files or subdirectories might be deleted as a result of this action. Many operating systems won't allow a directory to be deleted unless it is completely empty!

On the file side, choosing FILE1.TXT in the root directory gives us the opportunity to dump this file (see Figure 1). The first cluster position (offset 0x1A) in the FILE1.TXT directory entry (0x0002) points to where this file's data is stored. From the File_Size position (0x1C) in the FILE1.TXT directory entry, this file's length is 0x00000004. The first sector dump of cluster 0x0002 LBA 0x0280 shows the first four characters of this file to be "Test" (see Figure 5). This circuit's serial port is used as the output device for the dump operation. The file length determines how many characters will be sent. The characters are collected from every LBA (every sector of every cluster in the file's chain) until the appropriate number of characters have been transmitted. This is easy for FILE1.TXT because it has only four characters. However, the HALFDO~1.JPG file has multiple clusters. To dump this file, we begin with the first sector of cluster 0x000C, LBA 0x03C0 (0x000B × 0x20 + 0x0260), and send all 512 bytes. The LBA is incremented and all bytes of each sector are sent until the whole cluster (0x20 sectors) has been sent (0x100 bytes × 0x20 sectors = 0x2000 or 8,192 bytes). The FAT entry at word offset 0x000C is then interrogated to find out which cluster holds the next portion of data. An entire cluster's worth of data is sent (another 8,192 bytes.) The whole FAT thing is repeated until 2,644,245 bytes have been transmitted.

RING AROUND THE ROSIE

The last function, and the main reason for this project, is the logging of data from the serial port. If you have been following the processes up to this point, you should have a good understanding of how this is accomplished. The serial port has been implemented with output and input ring buffers. Each ring buffer has a head and a tail pointer. When the buffers are empty, the head pointer equals the tail pointer. Each pointer can point to any address of the buffer from its lowest address to its highest address (the buffer's length). The pointers are usually incremented and must be repositioned to the beginning of the buffer if they exceed the buffer's length—thus the term ring, or circular buffer. Once the ring buffers are implemented, you no longer have to deal with the

Address	HEX								ASCII	
	00	01	02	03	04	05	06	07	08	09
0910	1309	00FF	FFFF	FF00	0000	0000	210B	FE2E	!...
0920	2020	2020	2020	2020	2020	1000	0000	0000
0930	0000	0000	0000	0000	0004	0000	0000	002E
0940	2E20	2020	2020	2020	2020	1000	0000	0000
0950	0000	0000	0000	0000	0000	0000	0000	0054T
0960	574F	2020	2020	2020	2020	1000	0000	0000	WO
0970	0000	0000	0000	0000	0005	0000	0000	0000
0980	0000	0000	0000	0000	0000	0000	0000	0000
0990	0000	0000	0000	0000	0000	0000	0000	0000
09A0	0000	0000	0000	0000	0000	0000	0000	0000
09B0	0000	0000	0000	0000	0000	0000	0000	0000
09C0	0000	0000	0000	0000	0000	0000	0000	0000
09D0	0000	0000	0000	0000	0000	0000	0000	0000
09E0	0000	0000	0000	0000	0000	0000	0000	0000
09F0	0000	0000	0000	0000	0000	0000	0000	0000
0A00	0000	0000	0000	0000	0000	0000	0000	0000
0A10	0000	0000	0000	0000	0000	0000	0000	0000
0A20	0000	0000	0000	0000	0000	0000	0000	0000
0A30	0000	0000	0000	0000	0000	0000	0000	0000
0A40	0000	0000	0000	0000	0000	0000	0000	0000
0A50	0000	0000	0000	0000	0000	0000	0000	0000
0A60	0000	0000	0000	0000	0000	0000	0000	0000
0A70	0000	0000	0000	0000	0000	0000	0000	0000
0A80	0000	0000	0000	0000	0000	0000	0000	0000
0A90	0000	0000	0000	0000	0000	0000	0000	0000
0AA0	0000	0000	0000	0000	0000	0000	0000	0000
0AB0	0000	0000	0000	0000	0000	0000	0000	0000
0AC0	0000	0000	0000	0000	0000	0000	0000	0000
0AD0	0000	0000	0000	0000	0000	0000	0000	0000
0AE0	0000	0000	0000	0000	0000	0000	0000	0000
0AF0	0000	0000	0000	0000	0000	0000	0000	0000
0B00	0000	0000	0000	0000	0000	0000	0000	0000
0B10	0000	0000	0000	0000	0000	0000	0000	00EE

Figure 4—The root directory's subdirectory entry (ONE) points to cluster 0x00004, LBA 0x2C0. This dump shows how it is similar to the root directory but has two required entries, the "dot" entry pointing to its own cluster, and the "dot dot" entry pointing to a directory (one level up), in this case the root directory. This subdirectory also holds a new subdirectory (TWO).

serial port hardware directly, just the loading of the output buffer and unloading of the input buffer.

From the serial port side, any characters received cause an RX interrupt. The interrupt routine handles taking the received character and putting it into the input ring buffer. The routine may add only characters to the ring buffer via the buffer's head pointer. Received characters are placed at the head pointer (and the head pointer is incremented) only if there is room in the buffer. There is room until the head pointer + 1 equals the tail pointer. At this point, adding a character (and incrementing the head pointer) makes the head and tail pointers equal. This was previously defined as an empty buffer, so this would produce a buffer overrun condition and the buffer's data would be lost. One of two things must happen at this point, either the serial port must use flow control to stop the data from coming in or the data must be tossed out. This should not occur if the application can remove the data from the ring buffer and store it in the SD media faster than the data can be received.

On the serial output side, the output ring buffer will be

emptied via the output ring buffer's tail pointer. Unless the output ring buffer's tail pointer equals the head pointer, there is a character available for transmission. The TX interrupt routine is responsible for keeping the output ring buffer empty.

When the four characters of FILE1.TXT were dumped, the characters were placed into the output ring buffer using the buffer's head pointer. Because moving characters from the sector buffer to the output ring buffer will be fast, the application may stall waiting for the TX interrupt routine to empty the output ring buffer. So the dump time will be directly related to the data rate.

When logging data, assuming the data rate is sufficiently high compared to the data input rate, any bottleneck will come from the SD cards inability to write a block of data fast enough and get back for more without allowing the ring buffer to wrap. I thought I'd try logging a 0.5-MB file at a data rate of 19,200 bps for a test. I expected approximately 2,000 characters per second. I saw a sector write (lasting 35 ms) every 250 ms. That's four sectors, or 2,048 (i.e., 512×4) bytes per second. I pulled the SD card out and put it in my PC to check the file. It was the proper length at 567,408 bytes and viewed correctly. So, while I was in Windows Explorer, I used it to create an empty text file to try another test at a higher data rate.

I put the SD card back into my project board and repeated the test. I saw a sector write (lasting 35 ms) every 125 ms. Looking good! However, when I ended this logging session, the directory was trashed. (It viewed as if it had lots of garbage entries.) Hmm. It must have

run into timing issues. But wait, that would have caused a loss of data and not affected the directory. Hmm. The short story is the directory entry created using Windows Explorer didn't assign a FAT entry (so the FAT entry was zero) because an empty file has 0 bytes. When I began logging, I looked at the directory entry's FAT and assumed it had been assigned. (After all, that's what I do in this application.) Because a FAT entry of zero is used by the root directory, logging to it causes the root directory's sector to be overwritten, causing catastrophic results. With this incorrect assumption corrected, logging at 38,400 bps worked as expected.

SO MUCH MORE

While this project succeeds in performing the tasks required to explain

Address	HEX								ASCII	
	1309	00FF	FFFF	FF00	0000	0000	210B	FE54!..T
0910	6573	7400	0000	0000	0000	0000	0000	0000	est....
0920	0000	0000	0000	0000	0000	0000	0000	0000
0930	0000	0000	0000	0000	0000	0000	0000	0000
0940	0000	0000	0000	0000	0000	0000	0000	0000
0950	0000	0000	0000	0000	0000	0000	0000	0000
0960	0000	0000	0000	0000	0000	0000	0000	0000
0970	0000	0000	0000	0000	0000	0000	0000	0000
0980	0000	0000	0000	0000	0000	0000	0000	0000
0990	0000	0000	0000	0000	0000	0000	0000	0000
09A0	0000	0000	0000	0000	0000	0000	0000	0000
09B0	0000	0000	0000	0000	0000	0000	0000	0000
09C0	0000	0000	0000	0000	0000	0000	0000	0000
09D0	0000	0000	0000	0000	0000	0000	0000	0000
09E0	0000	0000	0000	0000	0000	0000	0000	0000
09F0	0000	0000	0000	0000	0000	0000	0000	0000
0A00	0000	0000	0000	0000	0000	0000	0000	0000
0A10	0000	0000	0000	0000	0000	0000	0000	0000
0A20	0000	0000	0000	0000	0000	0000	0000	0000
0A30	0000	0000	0000	0000	0000	0000	0000	0000
0A40	0000	0000	0000	0000	0000	0000	0000	0000
0A50	0000	0000	0000	0000	0000	0000	0000	0000
0A60	0000	0000	0000	0000	0000	0000	0000	0000
0A70	0000	0000	0000	0000	0000	0000	0000	0000
0A80	0000	0000	0000	0000	0000	0000	0000	0000
0A90	0000	0000	0000	0000	0000	0000	0000	0000
0AA0	0000	0000	0000	0000	0000	0000	0000	0000
0AB0	0000	0000	0000	0000	0000	0000	0000	0000
0AC0	0000	0000	0000	0000	0000	0000	0000	0000
0AD0	0000	0000	0000	0000	0000	0000	0000	0000
0AE0	0000	0000	0000	0000	0000	0000	0000	0000
0AF0	0000	0000	0000	0000	0000	0000	0000	0000
0B00	0000	0000	0000	0000	0000	0000	0000	0000
0B10	0000	0000	0000	0000	0000	0000	0000	000E

Figure 5—The minimum chunk that can be allocated to a file (or subdirectory) is one cluster. A cluster has 32 sectors associated with it. This file (FILE1.TXT) of only 4 bytes and any file, will have at least 32 sectors reserved for it.

how to use SD memory in a project, there is plenty more that can be discussed. The Microchip Technology PIC24FJ64GA002 has other useful hardware that you can explore. I purposely left time and date stamping out of my directory entry routines (other than making sure the entries were legal). This microcontroller has a hardware real-time clock that you can use to implement accurate time and date entries. You will also find a programmable-length CRC generator, which would make using CRCs a lot easier. I'll leave these and other enhancements for to you to experiment with. If you would like me to devote additional space to any of this, drop me an e-mail.

You'll find C routines implementing the FAT file system offered by many manufacturers. However, you won't learn much about it by just calling someone

else's code. I like to seize every opportunity to expand my knowledge base. Time constraints don't always allow this process, but I hope I've sparked your curiosity. Every so often, you should try to take this less traveled path. ☒

Jeff Bachiochi (pronounced BAH-key-AH-key) has been writing for Circuit Cellar since 1988. His background includes product design and manufacturing. You can reach him at jeff.bachiochi@imaginethatnow.com or at www.imaginethatnow.com.

SOURCE
PIC24FJ64GA002 Microcontroller
 Microchip Technology, Inc.
www.microchip.com

HandsOn Technology

<http://www.handsontec.com>

creativity for tomorrow's better living...

HandsOn Tech provides a multimedia and interactive platform for everyone interested in electronics. From beginner to diehard, from student to lecturer... Information, education, inspiration and entertainment. Analog and digital; practical and theoretical; software and hardware...



HandsOn Technology support Open Source Hardware(OSHW) Development Platform.

Learn : Design : Share

www.handsontec.com

