by Ivan Sham, William Hue, & Pete Rizun

Portable FAT Library for MCU Applications

The DOS FAT file system is the industry standard format for flash memory cards. You can use a microcontroller to read and write files on an SD/MMC flash memory card with the FAT system. Read on to learn how to build a portable FAT library for MCU applications.

Many new microcontroller applications require the ability to store large volumes of data. Transferring data between a portable device and a PC has become increasingly simple with the emergence of portable flash memory cards such as the Secure Digital (SD), MultiMediaCard (MMC), CompactFlash (CF), and xD Picture Cards. For the data to be easily readable on the PC, it must be stored using a file system supported by the operating system. The industry standard format for flash memory cards is the DOS file allocation table (FAT) file system. In this article, we'll show you how to use a microcontroller to read and write files on an SD or MMC flash memory card using the FAT system.

If you've ever used a USB memory key or attached a digital camera to your computer, then you've probably noticed that a new drive appears in Windows Explorer. You can drag and drop files to and from this drive. Low-cost flash memory card readers work the same way. Wouldn't it be useful to write files with your microcontroller that anyone can open on a PC with a standard card reader? No special software required! Similarly, wouldn't it be convenient to read files from a flash memory card that were written by a computer? In order to do so, you need to be able to interpret the FAT file system with your microcontroller.

The complete source code needed for using a FAT16-formatted SD/MMC with a Texas Instruments MSP430 microcontroller is posted on the *Circuit Cellar* ftp site. This way you don't have to write your software from scratch. At the end of this article, we'll tell you how to employ the software to write files that can be read with Windows Explorer and a standard flash memory card reader/writer. First, let's examine how to talk to the SD/MMC at the lowest level.

HARDWARE

SD and MMC flash memory cards both support the SPI transfer protocol and share an almost identical electrical interface. Connecting the flash memory card for operation in SPI mode to a microcontroller is similar to connecting a standard SPI device (see Figure 1). A bidirectional chip-select pin facilitates card detection. Pins 8 and 9 can be left open. To create an easily adaptable FAT decoding library, roughly 1 KB of RAM is required on the microcontroller.

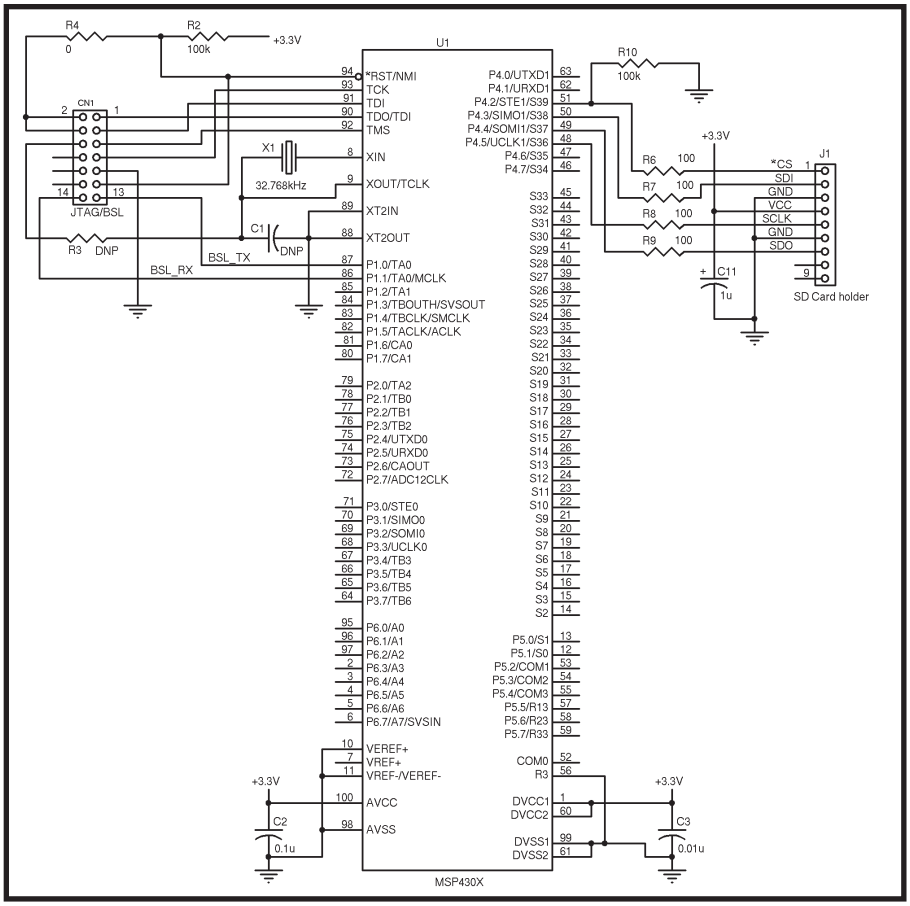

Figure 1—*Connecting an SD/MMC card to a microcontroller is almost as easy as connecting any other SPI device. The CS line serves as the SPI chip select, but it also can be polled to determine the presence of a flash memory card. Because SD cards are slightly thicker, the MMC will fit into an SD cardholder, but not vice versa. We used the Molex MLX54786 surface-mount SD cardholder.*

We chose the MSP430F449 because it has 2 KB of RAM, leaving a full 1 KB for stack and other variables. Photo 1 shows the latest project we completed with SD/MMC flash memory cards.

MEMORY CARD ACCESS

The low-level data transfer between the microcontroller and the SD/MMC is relatively straightforward. An SPI interface permits communication with only four physical connections (CS, SDI, SDO, and SCLK) besides power and ground. In SPI mode, the SD and MMC share an identical set of instructions.

To read or write data stored on a memory card, you must first perform the proper initializations. Because both the SD and MMC cards start in their native transfer modes at power-up, they must be placed in SPI mode. The initialization code needed to accomplish this is shown in Listing 1.

SD/MMC cards sample data input on the rising clock edge and set data output on the falling clock edge, so the microcontroller's SPI module should be configured accordingly. It's critical that the data input line is held high while waiting for the memory card to respond, as well as when the memory card is sending data to the microcontroller.

The first line of the initialization code disables the memory card by raising the active-low chip-select line (MEM_CS). The loop that follows ensures that the memory card has enough time to complete its internal power-up sequence. The `spi_put(byte x)` function clocks its argument out of the microcontroller and into the SDI pin of the memory card. It's important that the argument is 0xFF during this time in order to ensure the memory card powers up properly.

The chip-select line is then lowered and a 0x40 command (initialize card to the idle state) is sent with the arguments 0,0,0,0 and the CRC of 0x95. It's the state of the chip-select line (held low) during this reset procedure that places the memory card in SPI mode. The CRC for this command is precalculated. This is the only time the CRC value is critical, because the CRC is disabled after the card enters SPI mode. The `send_command(byte cmd, byte arg1, byte arg2, byte arg3, byte arg4, byte crc)` function simply transmits its arguments sequentially via six repeated calls to `spi_put(byte x)`.

Photo 1—*The board writes files to an SD card, which conveniently allowed us to store many megabytes of data, and then later transfer the data to a computer using readily available hardware. The MSP430 processor can write files to the flash memory card, which subsequently can be opened in Windows Explorer using a standard card reader. It can also read files from the flash memory card stored using a standard card reader/writer.*

The next line of the code calls `card_response(byte x)` to wait for an appropriate response from the memory card. This function repeatedly calls `spi_get()` and waits for the token specified by the argument before returning `true`. `False` is returned if a match is not received after a number of attempts. The expected token is 0x01 for the initialization command.

To get the card out of the idle state, the 0x41 command is sent repeatedly until the expected response is detected. If the card doesn't respond as expected after 255 tries, `initializeMemCard()` returns `false`, and the application can start anew.

After the expected response is detected, the chip-select line is set high and a dummy byte is sent, generating eight additional clock cycles to permit the memory card to complete its internal operations. The extra clock cycles are necessary for all commands.

We've given you an in-depth look at the initialization process because most events share a similar flow of events. First, lower the chip-select line. Second, clock out the command: command index (1 byte), arguments (4 bytes), and CRC (1 byte). Third, wait for a response byte from the memory card. Fourth, wait for the data token (for read operations), or clock out the data tokens (for write operations). Fifth, transmit or receive a block of data to or

Listing 1—*Initializing the memory card involves powering up the card properly and placing it in SPI mode. The state of the chip select line (held low) during this reset procedure places the card in SPI mode rather than its native Transfer mode.*

```
boolean initializeMemCard(void)
{
    MEM_CS = HIGH;
    for(i= 0; i < 10; i++)
    {
        spi_put(0xFF);
    }
    MEM_CS = LOW;
    send_command(0x40,0,0,0,0x95);
    if(card_response(0x01))
    {
        i = 255;
         do
        {
            send_command(0x41,0,0,0,0,0xFF);
            i—;
         } while (!card_response(0x00) && i > 0);
         MEM_CS = HIGH;
         spi_put(0xFF);
         if(i == 0)
        {
            return false;
        }
         return true;
    }
    return false;
}
```

from the memory card. Sixth, receive 2 bytes of error-checking information. Seventh, raise the chip-select line. Finally, generate eight clock cycles for the memory card to complete the operation.

The fifth and sixth steps were absent from the initialization routine, but they're the meat of read and write operations that we'll cover next. The memory card should automatically initialize to its 512-byte default read/write block size that matches the FAT sector size; nevertheless, it's good practice to ensure this using the set block length command. Table 1 shows all the byte codes for the SD/MMC commands needed to implement the FAT file system software. Although the memory card supports single- and multiple-block read operations, let's stick with single-block 512-byte reads (clock out the read block command, 0x51, with the byte address as the 4-byte argument).

Make sure the address argument for the read/write command is properly formatted for read/write operations. For example, if you want to read the 2,005 physical sector, the 4-byte address should be 0x00, 0x0F, 0xA8, 0x00 because the sectors are numbered starting with zero. Sector 2,004 has a 0x000FA800 byte address, which is determined in the following fashion: (2005–1) × 512 = 1,024,048 = 0x000FA800. Note that SD/MMC byte addresses are in big endian format.

The code in Listing 2 will read sector 2,004 in the memory card. It's similar to the initialization routine. The most notable difference is the presence of a second call to `card_response(byte x)`. The argument (0xFE) in the second call to this function is the data token. All data transfers with the memory card begin with this token. The 512 calls to `spi_get()` in the loop fill `buf` with the contents of sector 2,004. The two extra calls to `spi_get()` clock in the unused CRC bytes attached to every data block sent by the memory card.

Writing data is similar to reading data; the only difference is the direction of the data flow. In a read operation, the memory card provides the data token and the data. In a write operation, it's the microcontroller's responsibility to provide them. To write a block of data to the memory card, you have to send the Write Block command (0x58) with the properly formatted address, wait for the response byte, generate eight clock cycles, send the data token, and begin clocking out the data. Don't forget the 2 bytes of error code at the end of the data block. The code posted on the *Circuit Cellar* ftp site (WriteSDMMC.doc) writes the contents of the character array `buf` to sector 2,004 of the memory card.

Unlike with the read operation, you must make sure the memory card has completed its internal write operation after you clock out the data block. Checking the output from the memory card for a data-response byte can do this. The `checkWriteState()` function repeatedly calls `spi_get()` as it looks for the token 0x05. After the token 0x05 is detected, the completion of the write operation is signaled by the first nonzero byte detected by `spi_get()`.

Now that you know how to interface with the SD/MMC, let's look at how the FAT file system works. We'll provide you with a basic introduction to FAT. There's a wide range of literature describing the FAT file system, from its origins in the PCDOS and MSDOS operating systems to its current applications in Windows.

Command	Byte code	Response byte	Data token received	Data tokens transmitted
Initialize card to idle state	0x40	0x01	–	–
Bring card out of idle state	0x41	0x00	–	–
Read block	0x51	0x00	0xFE	–
Write block	0x58	0x00	–	8 clock cycle, 0xFE
Set block length	0x50	0x00	–	–

Table 1—*Although the SD and MMC flash cards respond to many commands, only four are required to implement our FAT library. It's good practice to set the card's block length to 512 to match the FAT sector size, but it isn't required because this should be the default value.*

FAT EXAMPLES

Let's first take a look at two examples of reading and writing files on a memory card. Assume we're working with a card formatted in FAT16, which uses 16-bit numbers to keep track of storage locations in the file system. Approximately 65,000 unique locations may exist in a FAT16-formatted disk.

A memory card is formatted like a hard disk with a single partition (the

Listing 2—*Reading a 512-byte sector involves sending the Read Block command with the address to read, and then generating the SPI clock signals to read the 512 bytes from the memory card, paying attention to the idiosyncrasies of the flash card.*

```
unsigned char buf[512];
boolean read_sector(void)
{
        unsigned short i;
        boolean retval = false;
        MEM_CS = LOW;
        send_command(0x51,0,0x0F,xA8,0,0xFF);
        if(card_response(0x00))
        {
          if(card_response(0xFE))
          {
              for(i = 0; i < 512; i++)
              {
                buf[i] = spi_get();
              }
              spi_get();
              spi_get();
              retval = true;
          }
          MEM_CS = HIGH;
          spi_put(0xFF);
        }
        return retval;
}
```

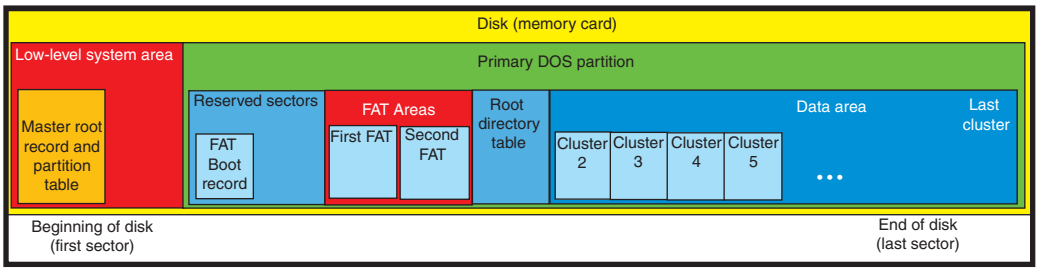

Figure 2—*In this memory map of a FAT16-formatted memory card, the data area is organized as groups of sectors called clusters. The FAT file system keeps track of how much storage is allocated to each file one cluster at a time.*

primary DOS partition). As you can see in Figure 2, the first sector (512 bytes) of the memory card contains the master boot record (MBR). The FAT partition (primary DOS partition) immediately follows it. The partition begins with some reserved sectors (the first of which is the boot record, which shouldn't be confused with the MBR), one or more FATs, and a root directory table. The data area follows the root directory; it's organized as groups of sectors called clusters. Storage space is allocated to files one cluster at a time.

Note that the MBR contains a small bootstrap loader program and the partition table. In a real hard disk, the bootstrap loader finds and runs the secondary bootstrap loader in the active partition's boot record. In a memory card, the MBR still contains the primary bootstrap program, but computers only use the partition table information.

READ A FILE

Consider the task of reading the file called hello.txt stored in the root directory and placing its contents ("Hello World!") in a character array. You can experiment with this example by typing "Hello World!" in a text editor and saving the file in a memory card's root directory. To read the file, you must first locate its entry in the root directory table located in the memory card after the FAT tables (see Figure 2).

The root directory table is organized like a spreadsheet (see Table 2). Each 32-byte row corresponds to a possible file entry. Some rows, however, might be blank (filled with 0 bytes) or refer to deleted files. The spreadsheet's columns provide information about the file. The first column represents the name. The second column represents the extension. (FAT16 uses 8.3 file names. The decimal point isn't recorded.) The penultimate column contains the number of the first cluster that contains the file's data. The final column contains the file size in bytes (little endian format). Other columns (bytes 11 through 25) represent various attributes and creation/modification dates.

To locate the hello.txt file, examine the name and extension columns in the root directory and look for a match. Because the root directory is just data stored in physical memory, you can read the root directory table into your microcontroller sector by sector, using the `read_sector()` function. The first sector in the FAT partition is the boot record, which includes parameters that tell you the sector where the root directory begins. Note that these parameters can be complicated.

Let's assume for now that you know which sectors contain the root directory table. You also know the length, in bytes, of each row in the table. You can easily imagine the process of writing a function to search the spreadsheet for hello.txt. In return you get the row number corresponding to the file entry. If a match isn't found in the sectors assigned to the root directory table, then the sought after file doesn't exist in the root directory.

Byte numbers	0–7	8–10	11–25	26–27	28–31
File 1 properties	name	extension	attrib., date	first cluster	size
File 2 properties	name	extension	attrib., date	first cluster	size
…	…	…	…	…	…
File n properties	name	extension	attrib., date	first cluster	size

Table 2—*The root directory table is organized like a spreadsheet. Each row corresponds to a file entry. The columns provide information about the file. The number of root directory entries (rows) may vary depending on the size of the disk. A common value is 512, in which case this table occupies 32 sectors.*

The original designers of the FAT file system kept things simple. The first characters in the names of deleted files are replaced with a special (non-alphanumeric) value. The first row that begins with a 0 byte (0x00) indicates that no subsequent rows have been used. Therefore, unless the root directory held its maximum number of files at some point, the search can stop without having to read to the last row of the table.

Let's assume that your search is positive and that you know the row corresponding to the file entry for hello.txt. You can read the 2 bytes from the first cluster column in little endian format to determine where the first chunk of information is located. You can read the size column to determine the file's size. To start reading the file contents, you must locate the first sector of the first cluster. The boot record also contains a parameter that tells you how many sectors are grouped together in a cluster.

Note that the number of sectors per cluster (the allocation size) is determined when a disk is formatted. Larger clusters allow smaller FAT tables and larger disks to be used within the 65,000-location limit. Nevertheless, they result in wasted space at the ends of files that don't use most of their last clusters.

The data area starts right after the root directory, so you can locate the first sector of the first cluster in the following way. If the first cluster column for hello.txt contains the number 107, the data in hello.txt begins at cluster 107 within the data area. (The first cluster in the data area is cluster number 2, because entries 0 and 1 in the FAT are reserved for holding information about the FAT table.) Now suppose your FAT file system has four sectors per cluster. This means the beginning of the file is located in the absolute sector number:

$$(\text{first sector of data area}) + [(\text{first cluster} - 2) \times (\text{sectors per cluster})]$$

The first sector of data area is computed with the parameter from the MBR to locate the beginning of the FAT partition, and then uses the FAT boot record's parameters to locate the beginning of the data area.

Returning to hello.txt, note that its entry size column reveals that the file is 12 bytes long. Thus, everything necessarily fits in the first cluster (and the first sector). Now the `read_sector()` function can read the file data into your microcontroller. All the bytes after the twelfth are disregarded.

Imagine modifying hello.txt with your PC so it contains 1,000 lines reading "Hello World!". The contents of this modified file might not fit in a single cluster. How can you determine where the rest of the file data is located? The answer is in the FAT. After reading the data from cluster 107, you would go to entry 107 in the FAT to determine the next cluster's location.

There can be more than one identical copy of the FAT (see Figure 2). The boot record contains parameters that tell you how many FATs are present, the sector length of each FAT, and where the first FAT starts relative to the boot record. Each FAT16 entry is a 16-bit number in little endian notation, so entry 107 is at the 214 and 215 bytes in the first sector of each FAT.

Multiple copies of the FAT are maintained to aid in the recovery of a corrupted file system. The FAT file system structure is simple, but not too robust. If a power failure or computer crash occurs while you're updating information in a FAT, you could lose the allocation chains of numerous files. Each of the FAT copies is updated separately, so only one FAT could get corrupted during a failure. Operating system utilities such as CHKDSK and SCANDISK attempt to minimize data loss by recovering as much information as possible from the uncorrupted copies of the FAT.

To find the next cluster in the file, read the appropriate sector (usually from the first copy of the FAT) into the microcontroller using the `read_sector()` function and then examine the appropriate bytes. Let's say that entry 107 in the FAT contains the number 489. This means the second block data for the file is sitting in cluster 489. The third block of data is sitting at the cluster number written in FAT entry 489, and so on. Repeat this process until you've pieced the entire file together. You'll know when to stop looking for more data, because the directory entry will tell you how many bytes the file contains. Moreover, an entry in the FAT that contains a value of 0xFFF8 through 0xFFFF means there aren't additional clusters associated with the file.

WRITE A FILE

As a second example, consider creating the goodbye.txt file in the root directory with the contents "Goodbye World!". First, you need to find a usable row in the root directory table to make a new entry. You can do this by searching for file names starting with a hexadecimal character, 0xE5 or 0x00, corresponding to deleted or unused entries, respectively. After you find a usable entry, write the file name with the extension, time of creation, and file attribute to the root directory table. The attribute field at byte 11 of a directory entry contains a number of bit flags, including one that indicates whether the entry is a file or subdirecto-

ry entry. When creating regular files, you must set the attribute field to 0x20.

A complication is that you also need to record the location of your file's first cluster in the table. Before doing so, you need to know where that should be. There are two cases you must consider. In the first, you are reusing an entry for a deleted file. The first cluster value and the entire FAT chain for the old (deleted) file is still intact. You can write over the data in the clusters in the existing FAT chain. Add clusters if the new file is longer than the old one.

To reduce disk I/O when a file is deleted, only the first character of its file name is replaced with 0xE5. Its directory entry and the FAT chain are left intact instead of following the chain and setting all its cluster entries to 0x0000 (unused). This is why files could be undeleted. It's also the reason why you could run out of space on a disk that's still reporting available free space if you're trying to create a file in a different subdirectory than one in which the deleted entries were located. Later operating system versions will search for the FAT chains belonging to deleted files when this occurs.

If you are populating a previously unused directory entry and you must find an unallocated cluster, unused clusters are signified by a zero entry in the file allocation table. Simply scan through the FAT (by reading it from our memory card one sector at a time) and look for a zero entry. After an entry is found, change its value to 0xFFFF (by writing the modified sector back to the memory card) to signify the end of the file, and then record this location in the root directory table.

With the location of the empty cluster assigned to the new goodbye.txt file, you can start writing its contents ("Goodbye World!"). You must write at least the first sector of the cluster. Place the string in the beginning of a 512-byte character array buffer and use the `write_sector()` function to send it to the memory card. It doesn't matter what the buffer contains after the last character in your string. We knew that our entire file fit within the first sector of one cluster, so we didn't need to write the remaining sectors in the cluster.

After adding the content to the file, update the file size listed in the directory table. It's good practice to update the access/modify time for the file as well. If a file has grown and it requires more than one cluster of storage space, a new empty cluster must be located in the file allocation table. The location of this new empty cluster is placed in the entry for the previous cluster in the FAT, and the FAT entry for the new last cluster is changed to 0xFFFF to indicate that it's now being used and that it's the end of the allocation chain. In this fashion, a FAT chain (and therefore a file) may grow to any length as long as there are clusters available.

SOFTWARE

The task of creating microcontroller software to read and write FAT-formatted disks can be daunting. The basic software package posted on the *Circuit Cellar* ftp site should allow you to start reading and writing files to the root directory of a memory card formatted in FAT16.

We based this article on FAT16 because it makes the code simple to understand. FAT32 is more complicated, but numerous books and papers have been published that address every version of FAT (FAT12, FAT16, and FAT32).

We wrote the software for the MSP430F449 (see Figure 1). You can easily port it to another platform. We ported our library to the PIC18F*xxxx*. We have a commercial version that works with FAT32, manipulates files with long file names, and performs directory operations. The FAT software package is based on two modules, HALayer and FATLIB, each of which consists of a .C (source) file and an .H (header) file. To write a program that can access an SD/MMC card, simply include the two header files and call the appropriate functions in your main program.

Listing 3 is a sample main program that reads the first 12 characters from the hello.txt file in the root directory. It prints the string to an LCD (presumably the ubiquitous "Hello World!"). The program then creates a goodbye.txt file and writes the "Goodbye World!" string. After the main program is complete, you should be able to insert the memory card into a standard memory card reader/writer on your computer. You'll see the goodbye.txt file in Windows Explorer. Verify its contents.

Listing 3—*The demo program illustrates the process of reading and writing to the SD/MMC card with our FAT library.*

```
int main(void)
{
        signed int stringSize;
        signed char handle;
        char stringBuf[100];
        cpu_init();     //Initialize the CPU clocks, etc.
        eint();         //Enable global interrupts (if required).
        fat_initialize();       //Initialize the FAT library.
    handle = fat_openRead("hello.txt");
    if (handle >= 0)
    {
              stringSize = fat_read(handle, stringBuf, 99);
              stringBuf[stringSize] = '\0';
              lcd_print(stringBuf);
              fat_close(handle);
    }
              handle = fat_openWrite("goodbye.txt");
    if (handle >= 0)
    {
              strcpy(stringBuf, "Goodbye World!");
              stringSize = strlen(stringBuf);
              fat_write(handle,stringBuf, stringSize);
              fat_flush();  //Optional.
              fat_close(handle);
    }
    while (1)
    {
                            //Stay here.
    }
}
```

TIME TO WORK

We'll continue to enhance our FATLIB with features as required by our customers. Now it's your turn to develop an exciting application for an SD or MMC memory card.

Ivan Sham is studying engineering physics at the University of British Columbia. Ivan expects to graduate with a bachelor's degree in April 2005. You may reach him at ivan sham@gmail.com.

William Hue owns HUE-Mobile Enterprises. He specializes in embedded systems, real-time control systems, industrial-grade systems, RF technology, communications systems, mixed signal designs, automotive technologies, and data acquisition systems. He holds B.A.Sc. and M.A.Sc. degrees in electrical and electronics engineering from Simon Fraser University. William may be reached at william@hue-mobile.com.

Pete Rizun designs and manufactures electro-mechanical devices. He has a B.A.Sc. in engineering physics from the University of British Columbia. Pete is currently a Ph.D. student at the University of Calgary, where he's working on an initiative in surgical robotics known as Project neuroArm. You may contact him at pete@rizun.com.

PROJECT FILES

To download the code, go to ftp.circuitcellar.com/pub/Circuit_Cellar/2005/176.

RESOURCES

J. Bachiochi, "SmartMedia File Storage," *Circuit Cellar* 143–144, June–July 2003.

FAT File systems and SD/MMC commands, www.ntfs.com/fat-systems.htm.

FAT resources, www.systemsmedic.com/SoftwareEdu2.htm; http://students.cs.byu.edu/~cs345ta/labs/winter04_specs/lab_fat_help.htm; www.seas.ucla.edu/classes/mkampe/cs111.fq04/docs/dos.html; www.extonpccouncil.org/Resources/EPCC_Tech_Talk/d003f32.htm.

Microsoft Corp., FAT32 File system specification, www.microsoft.com.

———*Microsoft MS-DOS Programmers Reference*, ver. 5, Microsoft Press, 1991.

MMC Specification, www.mmca.org.

P. Norton, *The Peter Norton Programmer's Guide to the IBM PC*, Microsoft Press, 1985.

M. Sargent and R. L. Shoemaker, *The IBM PC from the Inside Out*, Addison-Wesley, 1986.

A. Schulman, et al, *Undocumented DOS*, Addison-Wesley, 1990.

SD Card info, www.sandisk.com.

C. Siechert, *The Power of Running PC-DOS 3.3*, 2nd ed., Management Information Source, 1987.

SOURCES

MLX54786 SD memory card connector
Molex
www.molex.com

MSP430x44x Microcontroller
Texas Instruments
www.ti.com