USB Flash Drive Controller Using SPI

AN15484

Author: Ernie Buterbaugh
Associated Project: Yes
Associated Part Family: CY7C67300
Software Version: CY3663 EZ-Host Development Kit V1.1, CY4640
Associated Application Notes: None

Application Note Abstract

This Application Note describes using the EZ-Host™ USB Host to access a USB flash drive. An SPI bus is used as the interface to connect the EZ-Host to an embedded processor. All of the necessary commands to access the USB Flash drive including read and write file are described.

Introduction

USB Flash drives, also referred as thumb-drive, pen-drive, have become an important device for storing various amounts of data. Floppy disks and ZIP drives are a thing of the past because of the USB Flash drives. They are easy to use and nearly all computers with a USB port can read and write them.

But what about those systems that do not have a Windows or Linux OS and a USB host? There are many embedded systems that would like easy access to data and what could be easier or ubiquitous as the USB Flash drive? However, these embedded systems need a way to easily interface to the USB flash drive without adding excessive hardware and software.

SPI to USB Flash Drive

The EZ-Host is a programmable USB host controller and can be used to control any full speed USB peripheral with the appropriate firmware. There are two modes of operation for the device: coprocessor and standalone. In coprocessor mode, the EZ-Host provides the low level control on the USB interface and passes the data to a processor attached to one of its interfaces. This processor is responsible for the overall USB operation and the device drivers. In standalone mode, the EZ-Host is a self contained USB system and no external processor is required to support a USB device. This is the mode used for the USB Flash drive controller.

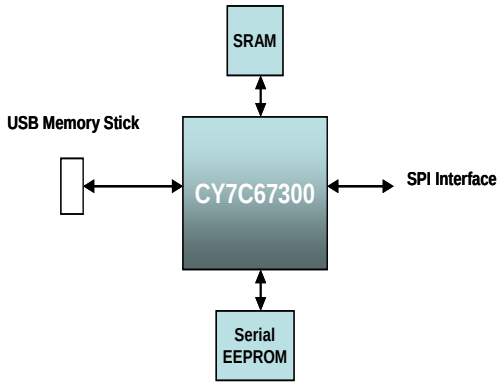
The SPI interface is simply the interface by which the embedded system reads and writes data to and from the flash drive using a low level protocol.

This application note discusses how to interface a SPI capable device to read and write to a USB Flash drive. The main component of this design is the CY7C67300, also known as the EZ-Host. Figure 1 shows a block diagram of the logic. Appendix A: Reference Schematic has a detailed schematic that can be used to implement this function. This design uses the following signals to the embedded processor:

- SPI MOSI
- SPI MISO
- SPI SS
- SPI Clock
- GPIO24
- GPIO25
- Reset

The four SPI signals are the standard SPI bus interface controls. GPIO24 and GPIO25 are outputs from the EZ-Host that signal the completion of various events. Reset is an input to the EZ-Host and allows the embedded processor to restart the code if necessary.

Figure 1. USB Memory Block Diagram



EZ-Host Firmware

As mentioned earlier, firmware is necessary for the EZ-Host to enumerate and control the USB flash drive. The code that is used by this application note is in the EZ-Host Mass Storage Class (MSC) Reference Design (CY4640) from the Cypress website (www.cypress.com). The code from version 1.1 of this kit is used with a few modifications. These changes are described in Appendix B: Changes in the Code.

About 56 KB of code is used to operate the USB Flash drive. This is larger than the 16 KB of RAM on the EZ-Host and hence the need for the external SRAM. Upon power-on, the RAM of the EZ-Host needs to be loaded with this code. This memory can be loaded automatically from an external serial EEPROM or it can be loaded directly across the SPI interface. An advantage of loading directly across the SPI is it eliminates the serial EEPROM and allows for easy code management. The advantage of having the serial EEPROM is the SPI interface control is simplified since it does not need to manage the booting process.

This application note uses the serial EEPROM as the method of loading the EZ-host memory and uses the SPI interface for data access.

EEPROM Code

The code that is programmed into the serial EEPROM is in a format called Link Control Protocol (LCP). This allows the code in the EEPROM to do more than simply load memory. It also allows for loading control registers, calling subroutines, and branching to an address in the code. You can find more information on LCP Commands in the *OTG-Host BIOS User Manual* which is part of the CY3663 developer's kit.

Upon power on, the BIOS in the EZ-Host reads GPIO30 (SCL) and GPIO31 (SDA) and if both are high (which they should be since these I^2C™ lines have pull up resistors), the EZ-Host executes the LCP commands from the EEPROM.

The LCP commands loads the RAM (both internal to the EZ-Host and the external SRAM) with the appropriate data. During this time GPIO25 is high. When the EEPROM is finished loading, GPIO25 is low. At this point, the memory is loaded with all of the code and the EZ-Host processor is waiting for a 'start' command from the SPI interface before it interfaces with the USB Flash drive.

Before moving on, a discussion on how to program the serial EEPROM could be helpful. There are the standard methods of programming the part using an external programmer before placing the part on the board. You can also program the part while it is on the board through USB port 2A on the EZ-Host chip. While in BIOS mode (that is before the EEPROM is programmed or anytime before the 'start' command is issued from the SPI interface) the EZ-Host enumerates as a device on this port. Simply attach this port to a PC and program the EEPROM with a utility called QTUI2C. This utility is part of the CY3663 developer's kit. When this kit is installed, a BASH environment is provided under the Cypress folder (Start, All Programs). The QTUI2C is executed in this BASH environment. The binary file that contains the code described is this application note is named:

MSC_EEPROM_scan_LCP_v2.bin

This file (or a later version) can be found with this application note. To load the EEPROM with this file, simply type the following in the BASH window after navigating to the project directory containing the .bin file:

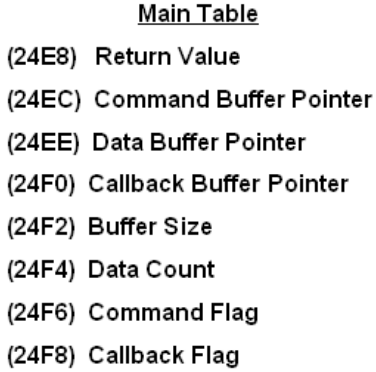
```
qtui2c MSC_EEPROM_scan_LCP_v2.bin f
```

System Architecture

Before discussing the details of the SPI communications, it is necessary to describe the architecture of the firmware that is operating on the EZ-Host.

There are a variety of buffers allocated in the EZ-Host memory that are used to pass commands, filenames, and data to and from the SPI device. By reading and writing these various memory locations, the SPI device has access to the USB Flash drive contents.

Figure 2. Main Parameter Table



There is one main table (Figure 2) that contains the pointers to the various buffers as well as status information. The address for the start of this table is in the EZ-Host memory at address 0x0310.

During the Initialization process, memory location 0x0310 is read to get the starting memory address of the Parameter Table. Memory location 0x310 is a fixed location; however, the pointer to the parameter table can change depending on how the EZ-Host MSC code is compiled. With code in the file *MSC_EEPROM_scan_LCP_v2.bin* which accompanies this application note, the value at 0x0310 is 0x24E8.

For reference, the addresses in parenthesis that are associated with the code in the file *MSC_EEPROM_scan_LCP_v2.bin* are shown in Figure 2 and throughout the remainder of this application note.

The following is a brief description of the values in this table.

Table 1. Main Parameters Table: Value, Size, and Description

Value	Size	Description
Return Value	32 bit	Gives status and information regarding an executed command. The values are dependant on the specific command.
Command Buffer Pointer	16 bit	Points to a location in memory where the various commands (e.g. open file) are loaded. For reference, this value is 0x24FA in v2 of the code.
Data Buffer Pointer	16 bit	Points to a location in memory where the various data values (e.g. filenames) are loaded. For reference, this value is 0x256A in v2 of the code.
Callback Buffer Pointer	16 bit	Points to a location in memory where the special return values are loaded. For reference, this value is 0x2D6A in v2 of the code.
Buffer Size	16 bit	Specifies the size of memory allocated in the EZ-Host for each of the three buffers. This value is 0x0800 in v2 of the code.
Data Count	16 bit	Indicates the maximum number of bytes in the current transaction.
Command Flag	16 bit	Indicates the command buffer has a valid command loaded and the EZ-Host should process the command.
Callback Flag	16 bit	Indicates that some event (e.g. USB Flash drive was removed) occurred that needs servicing.

In order to read and write the USB Flash drive, it is simply a matter of reading and writing the various buffers and flags.

SPI Interface

The necessary commands to control the USB flash drive can be broken down into four elements:

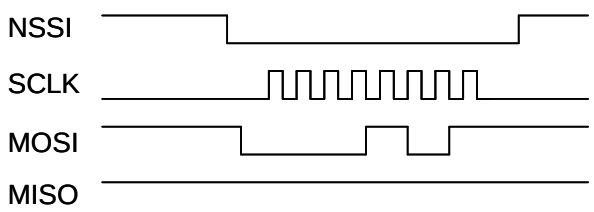
- SPI signals
- Read/Write Memory
- Disk Operations
- High Level Functions

This structure makes it easier to describe, and also follows how one would write the interface code on an embedded processor for the SPI master.

SPI Signals

Timing on the SPI interface uses the standard SPI bus signals: MOSI, MISO, SS, and SCLK. Figure 3 shows the timing waveform for a SPI transaction. Note that SS must toggle for every byte transferred. When sending and receiving data, the most significant bit (MSB) is the first bit clocked. The data sent from the SPI Master to the EZ-Host slave is 0x09 in Figure 3.

Figure 3. SPI Byte Transfer



The maximum SPI clock rate is 2 MHz. It is recommended that a small delay be placed between the de-assertion of Slave Select of one byte and the assertion of Slave Select for the next byte in order to allow the SPI slave on the EZ-Host to recognize and capture the incoming byte.

Read Memory

The read memory protocol allows the SPI master to read any memory in the EZ-Host. It contains a structure to specify the address and returned data. It also provides an acknowledgement control and checksum word to ensure data integrity. The following shows the structure for Reading Memory:

```
MOSI: 10 00 <Addr> <Len> 00 00
MISO: 18 00 <Addr> <Len> <CS> <Data>
MOSI: 18 00 <Addr> <Len> <CS>
```

In the first transaction where the SPI master is sending out data, the 16 bit command 0x0010 to read memory is sent. Note that the least significant byte (LSB) is sent first and then the most significant byte (MSB) of each 16 bit word on the SPI bus is sent. The next two bytes are the address of the memory location to be read. The two byte LEN field specifies how many bytes are to be returned. The address is automatically incremented and therefore the LEN can be any even number of bytes. And finally two bytes of 00's are sent to make the complete transaction eight bytes long.

Before the master can begin reading the data back from the EZ-Host slave, it must first wait for GPIO24 to pulse. This signals that the buffer has been read and the EZ-Host is ready to return data.

In the second transaction, the EZ-Host slave returns a read acknowledge of 0x0018 followed by the same Addr and Len as provided by the master. The next two bytes contain the checksum of the data being returned. The data bytes follow the checksum. The number of data bytes transferred is equal to the value specified by the Len field.

Again, the SPI Master must wait for GPIO24 to pulse before sending the final transaction.

The last transaction is simply an acknowledgement by the SPI Master back to the EZ-Host slave that the data was received. Note that these are the same eight bytes that the EZ-Host previously sent without the data bytes.

Here is a specific example where two bytes of memory address 0x310 is read and 0x24E8 is the value at that location. Note the byte ordering.

```
MOSI: 10 00 10 03 02 00 00 00
MISO: 18 00 10 03 02 00 17 DB E8 24
MOSI: 18 00 10 03 02 00 17 DB
```

The checksum that is inserted is a 1's complement form. The algorithm used adds each of the 16 bit values to form a 32 bit result. The upper 16 bits of the 32 bit result is added to the lower 16 bits to form a new 16 bit value. This value is then inverted. The C code used to implement this algorithm is in Appendix C: Checksum Algorithm Implemented in C.

Write Memory

The write memory protocol allows the SPI master to write any memory in the EZ-Host. It contains a structure not only to specify the address and the data, but it also provides an acknowledgement control and checksum word to ensure data integrity just like the read memory operation. The following shows the structure for Writing Memory:

```
MOSI: 20 00 <Addr> <Len> <CS>
MISO: 28 00 <Addr> <Len> <CS>
MOSI: <Data>
MISO: 28 00 <Addr> <Len> <CS>
```

In the first transaction where the SPI master is sending out data, the 16 bit command 0x0020 to write memory is sent. Again, note that the LSB is sent first and then the MSB of each 16 bit word on the SPI bus. The next two bytes are the address of the memory location to write. The LEN field specifies how many bytes are to be written. And in this case a 16 bit checksum is sent. As in the read, the checksum covers the data only and follows the same algorithm as the read.

The address is automatically incremented and therefore the LEN can be any even number of bytes. This transaction is always eight bytes long.

Before the master can begin reading the acknowledge transaction from the EZ-Host slave, it must first wait for GPIO24 to pulse. This signals that the buffer has been read and the EZ-Host is ready to return data.

In the second transaction, the EZ-Host slave returns a write acknowledge of 0x0028, followed by the same Addr, Len and CS as provided by the master.

Again, the SPI Master must wait for GPIO24 to pulse before sending the next transaction.

The SPI Master can now send the data bytes to the EZ-Host Slave. The number of bytes must equal the length specified in the preceding transactions.

Again, the SPI Master must wait for GPIO24 to pulse before sending the next transaction.

The last transaction is simply an acknowledgement by the EZ-Host slave back to the SPI Master that the data was received. Note that these are the same eight bytes that the EZ-Host previously sent.

Here is a specific example where two bytes of memory address 0x24FA is written with 0xC3B6. Again note the byte ordering.

```
MOSI: 28 00 FA 24 02 00 49 3C
MISO: 28 00 FA 24 02 00 49 3C
MOSI: B6 C3
MISO: 28 00 FA 24 02 00 49 3C
```

Disk Operations

There are a variety of commands used to control the USB Flash drive that the EZ-Host supports, and each follows a similar pattern. First an OPCODE and a few parameters are written to the Command Buffer. Then depending on the command, data may be written to the Data Buffer. The Command Flag is set to tell the EZ-Host to go execute that command. When the EZ-Host is finished processing the command, GPIO25 toggles. This tells the SPI Master that it can read the status bytes and data buffers depending on the particular command.

Each of the operations is described in detail following this example operation. This example executes the Show Current Directory operation. The directory name that is returned is capsen~1.

First write to the Command Buffer an OPCODE for Show Current Directory and its associated parameters. From the previous discussion, the Command Buffer starts at location 0x24FA in memory (this value is from the Main Table). The OPCODEs and parameters are described later. For the Show Current Directory we write these six bytes:

```
24FA <- F00D
24FC <- 0004
24FE <- 0104
```

This operation is performed by writing two bytes to the three respective memory locations, or simply, write six bytes to address 0x24FA.

Next, write the command flag to indicate the command is ready to be executed.

```
24F6 <- C3B6
```

After writing this command, the SPI Master must wait for GPIO25 to trigger before reading the status.

The SPI Master then reads the status word (Return Value) to ensure that it is 0000 indicating success.

```
24E8 -> 0000
```

Then read the number of bytes being returned as indicated in the Data Count (in the Main Table). In this specific case, 10 bytes are returned:

```
24F4 -> 000A
```

Note that these can be two separate memory reads but since they are in the same table, you could have read fourteen bytes starting at location 24E8 to get all of the data.

Since there are 10 (0x0A) bytes being returned, read the Data Buffer to get the character string of the directory name. Again with v2 of this code, the Data Buffer starts at address 0x256A and therefore, read the 10 bytes starting at this location:

```
256A -> 632F 7061 6573 7E6E 0031
```

Notice the several items in the contents of the buffer. First, the data bytes are swapped again – 2F ('/') is really the first character of the string followed by 63 ("C") and then 61 ("A"). The string is terminated with a null – 00. Therefore the buffer returns what we had expected:

```
/CAPSEN~1
```

File Operation Format

The following is a list of each OPERATION and the expected data. For each of the commands, various buffers and flags are set and then the function is executed. After execution, various buffers and flags are read. It is recommended to test the Callback Flag after every File Operation to determine if the USB device has been detached. The section titled USB Insertion and Removal discuses the Callback operation.

Note that only 8.3 filenames are supported. Microsoft long filenames extensions are not supported.

SHOW CURRENT DIRECTORY

This operation returns the full path of the current working directory. If the current working directory is the root, a slash (/) is returned. If the directory is something other than the root, the string contains the directory name.

Setup Parameters	
Command Buffer	F00D, 0004, 0104
Command Flag	C3B6
Returned Values	
Return Value	0000 = OK
Data Count	Number of bytes in Data Buffer
Data Buffer	<String of directory name><null terminated>

OPEN FILE

This operation opens the specified file for either reading or writing.

Setup Parameters	
Command Buffer	F000, 0000, 00nn nn equals the number of bytes of the filename including nulls. This must be an even number of bytes and therefore there can be one or two bytes of nulls.
Data Buffer	<String of filename><null terminated>
Command Flag	C3B6
Returned Values	
Return Value	FFFF = file not found; otherwise Return Value is the 16 bit File Handle.

CREATE FILE

The Create File operation creates the specified file. If the filename already exists, it is opened and truncated.

Setup Parameters	
Command Buffer	F005, 0000, 00nn nn equals the number of bytes of the filename including nulls. This must be an even number of bytes and therefore there can be one or two bytes of nulls.
Data Buffer	<String of filename><null terminated>
Command Flag	C3B6
Returned Values	
Return Value	FFFF = file not created; otherwise Return Value is the 16 bit File Handle.

ACCESS FILE

The Access File operation, when used with files, determines whether the file exists and can be accessed. When used with directories, it determines only whether the specified directory exists.

Setup Parameters	
Command Buffer	F008, 0000, 00nn nn equals the number of bytes of the filename including nulls. This must be an even number of bytes and therefore there can be one or two bytes of nulls.
Data Buffer	<String of filename><null terminated>
Command Flag	C3B6
Returned Values	
Return Value	FFFF = file not open; otherwise Return Value is the 16 bit File Handle.

CLOSE FILE

This operation closes the specified file.

Setup Parameters	
Command Buffer	F001, 0000
Data Buffer	File Handle
Command Flag	C3B6
Returned Values	
Return Value	0000 = OK

CHANGE DIRECTORY

This Change Directory operation changes the current working directory to the specified directory. The new directory must already exist.

Setup Parameters	
Command Buffer	F00C, 0000
Data Buffer	<String of directory name><null terminated>
Command Flag	C3B6
Returned Values	
Return Value	0000 = OK

REMOVE DIRECTORY

This Remove Directory operation deletes a directory with the specified name. The directory must be empty and must not be the current directory or the root directory.

Setup Parameters	
Command Buffer	F00F, 0000
Data Buffer	<String of directory name><null terminated>
Command Flag	C3B6
Returned Values	
Return Value	0000 = OK

MAKE DIRECTORY

This Make Directory operation creates a directory with the specified name.

Setup Parameters	
Command Buffer	F00E, 0000
Data Buffer	<String of directory name><null terminated>
Command Flag	C3B6
Returned Values	
Return Value	0000 = OK

REMOVE FILE

The Remove File operation erases the specified file. All handles for the specified file must be closed before it can be deleted.

Setup Parameters	
Command Buffer	F006, 0000
Data Buffer	<String of filename><null terminated>
Command Flag	C3B6
Returned Values	
Return Value	0000 = OK

RENAME FILE or DIRECTORY

The Rename File operation renames the specified file or directory. The old name must be an existing file or directory. The new name must not be the name of an existing file or directory.

Setup Parameters	
Command Buffer	F007, 0000, 00nn nn equals the number of bytes of the old filename including nulls. This must be an even number of bytes and therefore there can be one or two bytes of nulls.
Data Buffer	<String of old filename><null terminated><String of new filename><null terminated>
Command Flag	C3B6
Returned Values	
Return Value	0000 = OK

FIND FIRST

This Find First operation returns file information about the first instance of a file that matches the filename specified. The filename may include wildcards characters * or ?.

Setup Parameters	
Command Buffer	F016, 0004, <filename><null terminated>
Command Flag	C3B6
Returned Values	
Return Value	FFFF = file not found, otherwise Return Value is the 16 bit File Handle.
Data Count	Number of bytes returned in Data Buffer
Data Buffer	<String of filename><null terminated>

Note: The string name of the file or directory starts on the 13th byte in the Data Buffer and is null terminated.

FIND NEXT

This Find Next operation returns file information about the next instance of a file that matches the file specified. (Used in conjunction with Find First.)

Setup Parameters	
Command Buffer	F017, 0004, <file handle>
Command Flag	C3B6
Returned Values	
Return Value	FFFF = file not found; Filehandle if file found.
Data Count	Number of bytes returned in Data Buffer

Note: The string name of the file or directory starts on the 13th byte in the Data Buffer and is null terminated.

FIND CLOSE

The Find Close operation closes the specified handle and releases associated resources.

Setup Parameters	
Command Buffer	F018, 0000
Data Buffer	<file handle>
Command Flag	C3B6
Returned Values	
Return Value	0000 = OK.

END OF FILE

End Of File operation determines whether the end of the referenced file has been reached.

Setup Parameters	
Command Buffer	F004, 0000
Data Buffer	<file handle>
Command Flag	C3B6
Returned Values	
Return Value	0000 = Not at EOF or 0001 = EOF

READ FILE

This operation reads a maximum number of bytes (specified by Data Count) into the Data Buffer from the referenced file. The read operation begins at the current position of the file pointer. After the read, the file pointer points to the next unread character.

Setup Parameters	
Command Buffer	F002, 0008, 0000, 0000, <file handle>
Data Count	Max Number of bytes in Data Buffer
Command Flag	C3B6
Returned Values	
Return Value	Number of bytes returned in Data Buffer
Data Buffer	<data from file>

WRITE FILE

This operation writes the number of bytes (specified by Data Count) from the Data Buffer to the referenced file. The write operation begins at the current position of the file pointer. After the write operation completes, the file pointer increments by the number of bytes actually written to the file.

Setup Parameters	
Command Buffer	F003, 0008, 0000, 0000, <file handle>
Data Count	Max Number of bytes in Data Buffer
Data Buffer	<data for file>
Returned Values	
Return Value	Number of bytes written to file

USB Insertion and Removal

When the USB Flash drive is inserted or removed, the GPIO25 signal pulses. When this occurs, the SPI Master needs to read the Callback Flag in the Main Parameter Table. (The address is 0x24F8 in v2 of the code.) If this value is non zero, then a change of status has occurred.

The first step is to clear the Callback Flag by writing 0x0000 to that memory location. The second step is to read the Callback Buffer (The address of the buffer is 0x2D6A for v2 of the code.) Twenty six bytes of the Callback Buffer need to be read. Bytes 25 and 26 contain a value of either 0x0001 (USB Device Inserted) or 0x0002 (USB Device Unplugged).

If a USB Device was inserted, the first eight bytes of the Callback Buffer contains the Volume Label string. Here is an example of the CallBack Buffer Contents:

Callback Buffer:

```
7375, 3062, 003A, 0000, 0000, 0001, BA60,
0020, 0000, 0000, 0000, CF58, 0001
```

In this case bytes 25 and 26 contain a 0001 and therefore indicate the USB device was inserted. The volume label from the first eight bytes shows the name of: “usb0:”.

The other bytes in the buffer provide additional disk information.

High Level Functions

Many of the Disk Operations provide meaningful results (e.g. SHOW CURRENT DIRECTORY) as a standalone function. However, it is useful to link several of the operations together to create higher level functions such as reading or writing a file.

The High Level Functions section describes how to interact with the USB flash drive as you might from a DOS or Linux prompt. They allow you to view the contents of a directory, read a file, create a directory, and many more functions. Some simply use a single Disk Operation and others loop through several operations. This application note describes various High Level Functions most often used and similar to those found in an OS command line environment.

The following describes the list directory contents command (i.e. LS for Linux, DIR for DOS) to illustrate the process that is used. In order to execute this function, several of the low level disk operations are called:

- Find First
- Find Next
- Find Close

The first operation is the Find First with a filename of *.*. From the description given above, it is known that *.* is part of the command string and sent to the EZ-Host. After the command is executed, the return code is read and if not 0xFFFF, the file exists. Of course in this case, wildcards of *.* are being sent and therefore, all files match. If you only want to list files with an extension of .txt, you could have pass the filename *.txt to list only those type of files.

If files do exist, the return code is the file handle which is used in subsequent operations. The name of the first file or directory is also returned in the Data. The amount of data in the Data Buffer is indicated by the Data Count byte. Note that the actual name of the file or directory starts on the 13th byte and is null terminated.

Now that the first file/directory name has been retrieved, loop on the next operation. Use FIND NEXT to retrieve the names of all the other names.

Pass the filehandle to the FIND NEXT operation and then execute. If the return code is 0xFFFF then no more file/directory names exist and we proceed to the FIND CLOSE operation.

If the return code is equal to the filehandle, then a new name is in the Data Buffer. As with the FIND FIRST command, read the Data Count parameter for the number of valid bytes in the buffer. Again, the actual name of the file or directory starts on the 13th byte and is null terminated.

Repeat the FIND NEXT operation until it returns with a 0xFFFF status code indicating no more names.

And finally to end this function a FIND CLOSE operation is performed. This is accomplished by setting up the command buffer with the FIND CLOSE opcodes, placing the filehandle in the Data Buffer and then executing. The Return Code should be a 0x0000 when complete.

High Level Functions are described in the following format. The first parameters in parentheses are the relevant returned data from the File Operation. The File Operation command is in all caps. And the data provided to the File Operation are the last parameters in parentheses. The operation just described is noted as follows.

LIST Contents

```
(filehandle, name) FIND FIRST (*.*)
(status, name) FIND NEXT (filehandle)
(status) FIND CLOSE (filehandle)
```

Summary: Get filehandle and the first name using the FIND FIRST command. Continue with FIND NEXT commands until no names are found. End the operation with the FIND CLOSE command.

The following describes the most often used High Level Functions. Other functions can be created using the various disk operations in the same manner. This application note describes these functions:

- Make Directory
- Remove Directory
- Rename
- Create File
- Remove File
- Show Working Directory
- List Directory
- Change Directory
- Read File
- Write File

Show Working Directory (CWD)
(status, name) SHOW CURRENT DIRECTORY

Summary: Simply execute the SHOW CURRENT DIRECTORY disk operation. The name of the directory is returned in the data buffer starting at byte 0 and null terminated. See Appendix D: SPI Initialization Sequence for an example SPI sequence of this function.

Make Directory (MKDIR)
```
(status) MAKE DIRECTORY (name)
```
Summary: Simply execute the MAKE DIRECTORY disk operation. The name of the directory is placed in the data buffer. Status returns with success or failure.

Remove Directory (RMDIR)
```
(status) REMOVE DIRECTORY (name)
```
Summary: Execute the REMOVE DIRECTORY disk operation. The name of the directory is placed in the data buffer. Status returns with success or failure.

Rename File (RENAME)
```
(status) RENAME FILE (old filename, new
filename)
```
Summary: Execute the RENAME FILE disk operation. The name of the old filename and the new filename are placed in the data buffer. Nulls separate the two names and the string is null terminated. Status returns with success or failure.

Change Directory (CD)
```
(status) CHANGE DIRECTORY (name)
```
Summary: Execute the CHANGE DIRECTORY disk operation. The name of the directory is placed in the data buffer. Status returns with success or failure.

Read File (CAT)
```
(filehandle) OPEN FILE (filename)
(status) END OF FILE (filehandle)
(status, data) READ FILE (filehandle)
(status) CLOSE FILE (filehandle)
```
Summary: Open the filename by executing the OPEN FILE disk operation. If successful, a filehandle is returned. If a file handle is returned, test the file status by executing the END OF FILE command. Status indicates end (0x0001) or not at end (0x0000) of file. If at the end of the file, start the CLOSE FILE operation. If not end of file, continue with READ FILE. Execute the READ FILE and the status indicates how many bytes are being returned in the Data Buffer. After the data is retrieved, continue to loop through the END OF FILE and READ FILE operations until an End Of File is reached or the desired number of bytes are captured. Finish by executing the CLOSE FILE operation.

Write File (Write)
```
(filehandle) CREATE FILE (filename)
(status) WRITE FILE (filehandle, data)
(status) CLOSE FILE (filehandle)
```
Summary: Create the filename by executing the CREATE FILE disk operation. If successful, a filehandle is returned. Note that you may perform an ACCESS FILE or OPEN FILE command to see if the file does exist in which case the CREATE FILE is not necessary. Next write a block of data. This can be as much as 2K bytes but is generally dictated by the size of the buffer in the controlling processor. The status indicates successful writes. Finish by executing the CLOSE FILE operation.

Remove File (RM)
```
(filehandle, name) FIND FIRST (filename)
(status) REMOVE FILE (filehandle)
(status, name) FIND NEXT (filehandle)
(status) FIND CLOSE (filehandle)
```
Summary: Find the first occurrence of the filename. Note that the filename can be the full name or a name with wildcards and hence the need to search the directory for other occurrences. If found, remove the file by providing the returned filename and executing the REMOVE FILE operation. Next perform a FIND NEXT operation to determine if another file exists. If not, proceed to FIND CLOSE. If a file does exist, execute REMOVE FILE with the newly returned file name from the Data Buffer. Continue the loop of FIND NEXT and REMOVE FILE until FIND NEXT returns no more names. Finally execute FIND CLOSE to finish.

Create File (CREATE)
```
(filehandle) CREATE FILE (filename)
(status) CLOSE FILE (filehandle)
```
Summary: Create a file whose name is placed in the Data Buffer using the CREATE FILE Operation. The CREATE FILE returns with the 16 bit file handle if successful. Close the file using the CLOSE FILE operation and pass the file handle from the CREATE FILE operation.

Initialization Sequence

Now that all of the commands and operations are known, the last item of importance is how to start the EZ-Host Mass Storage Code after the EEPROM has loaded the memory. As mentioned earlier, the code that is with this application note is booted into the EZ-Host memory on power on. The EZ-Host then waits for a command sequence to start.

GPIO25 is held low after the serial EEPROM contents is loaded into the SRAM. Since this GPIO can glitch at

power on or at reset, ensure that GPIO25 is low for at least 100 µs before sending the initialization sequence. If the EZ-Host is controlled by a microprocessor, it is recommended to issue a reset to the EZ-Host with the microprocessor and then monitor GPIO25. This allows the ability to re-try should something fail during the initialization sequence.

Once the EEPROM is loaded and the EZ-Host signals it is ready, send an LCP command to instruct the EZ-Host to start the code. The LCP command that is sent is a JUMP to 0x04A4. This is the starting location in memory of the EZ-Host program.

The SPI master sends the following bytes to the EZ-Host:

```
MOSI: 04 CE A4 04 00 00 00 00
```

Note that the documentation for the EZ-Host BIOS indicates that the SPI Master should continuously read MISO until the 0x0FED response is returned before proceeding. However, since this code re-initializes the SPI hardware, the 0x0FED response is delayed until the reading of the next status.

Upon successful completion of this command, GPIO25 pulses high and then back to low. The code in the EZ-Host is now operational.

Before starting, it is recommended to read a memory location to clear out any extraneous status on the SPI bus (i.e. the 0x0FED from the JUMP command). Reading two bytes from memory location 0x310 is sufficient. Instead of getting back 10 bytes of data, you should read 12 bytes of data. The first two bytes is the status from the LCP JUMP instruction and should be 0x0FED. The SPI sequence in Appendix D shows this initialization.

At this point, read memory at 0x310 and get its value. This is the pointer to the Main Table. As mentioned earlier, the binary code (*MSC_EEPROM_scan_LCP_v2.bin*) that is with this application note should return a value of 0x24E8. With this pointer now known, you can determine the location of all of the status registers and each of the memory buffer locations.

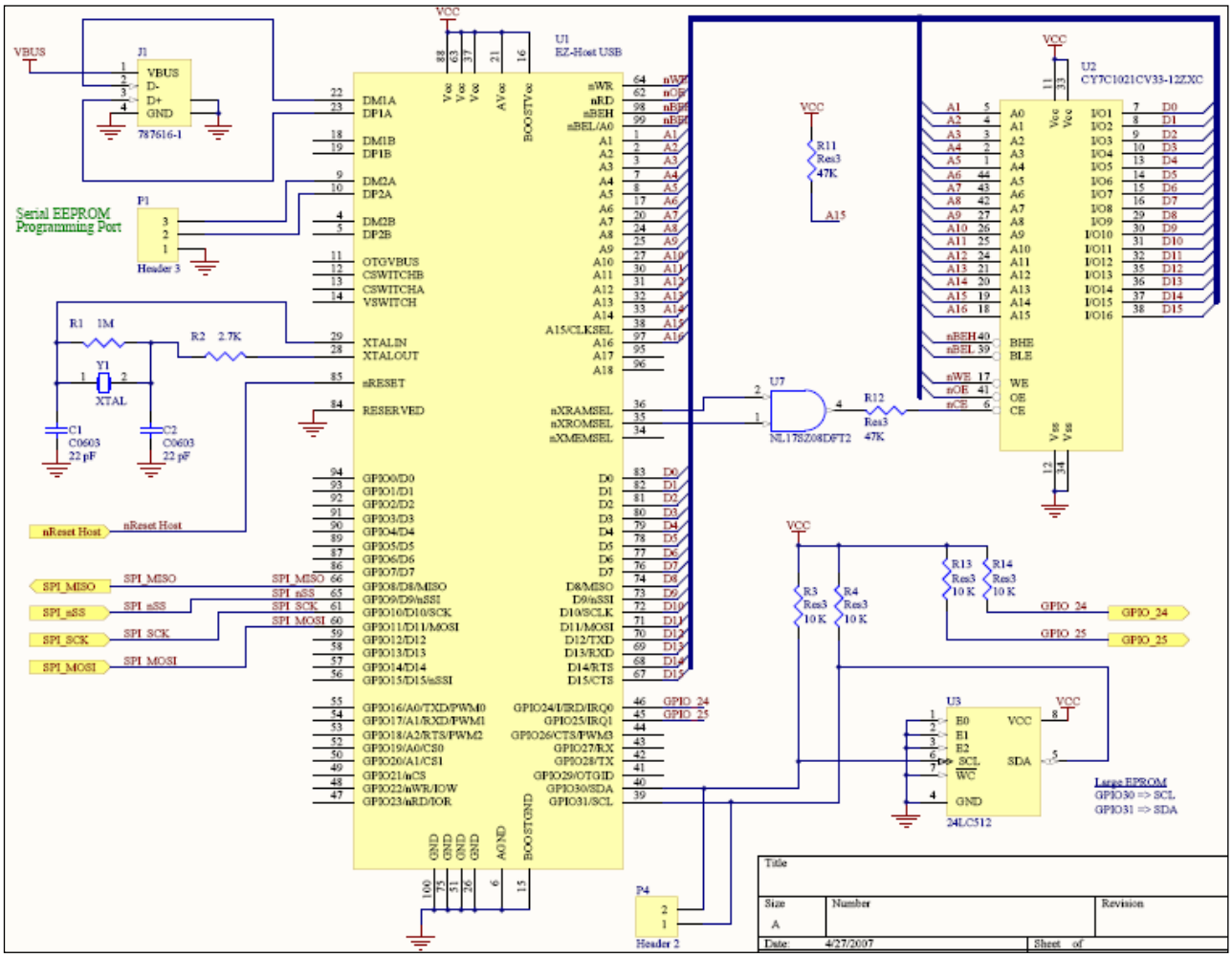
The last step is to read the Command Flag in the Main Table. The value must be 0x55AA. If it is not, then the code is not operating correctly. If it is correct, write a 0x0000 to the Command Flag to clear the flag.

Conclusion

USB Flash drives have gained a tremendous amount of popularity. Most people have them and they are easily accessible. All PCs with USB ports support the Flash drive making them very easy to read and write. By using the EZ-Host with the Mass Storage Code firmware, your embedded processor can now easily take advantage of USB Flash drives too. By using a SPI interface and the appropriate commands, reading and writing data is at your command.

Appendix A: Reference Schematic

Figure 4. Reference Schematic for SPI Bus Interface



Appendix B: Changes in the Code

The code that is included in the CY4640 Version 1.1 Mass Storage Class Reference Design was used as the baseline code for this application note. There are a few changes to this code to allow support for loading the EEPROM and controlling the SPI interface. A few corrections to the algorithms were also made. The following details the changes made to the source in order to create the file *MSC_EEPROM_scan_LCP_v2.bin*.

comm._driver.c (in MSC_API subdirectory)

The CY4640 Version 1.1 Checksum routine is incorrect. It is replaced with the code as shown in Appendix C: Checksum Algorithm Implemented in C.

app.c (in MSC_API subdirectory)

Added support for the LCP code in the EEPROM to branch to a known place in code. This code enables the SPI hardware, signals that the code load is complete, and then branches to BIOS to wait for LCP commands from the SPI interface. This code is located at 0x4E26 in EZ-Host memory. This is an added callable code segment placed directly before app_pre_init. The last LCP command in the EEPROM code JUMPS to this location.

```
/* This function is also called by the EEPROM LCP Functions */
/* After the EEPROM is loaded via I2C, the last command jumps here */
/* Then the memory loads and waits for an LCP command */
/* to jump to 04A4 ----     EWB  */

void EEPROMLoadReady(void)
{
   __asm("call 0xE998");                    /* call SPI_GInit */

   *(volatile uint16*)(0xc028)|= 0x0200;    /* Lower GPIO25 */

   __asm("jmp 0xE0A2");                     /* jmp to BIOS exec */
}
```

app.c (in MSC_API subdirectory)

The raise_interrupt routine has been changed to have added delay for the GPIO25 interrupt line which ensures a true rise and fall pulse. It also uses drive high and drive low writes to the GPIO port instead of an XOR function.

```
/*****************************************************************
 * FUNCTION    : raise_interrupt
 * PURPOSE     : Raise and lower GPIO25
 *
 * DESCRIPTION : Helper function for signaling external processor
 * PARAMS      :
 *
 * RETURNS     : void
 *****************************************************************/
void raise_interrupt(void)
{

 /*Raise interrupt for external processor.Raise GPIO25 */

   *(volatile uint16*)(0xc028)&= ~0x0200;   /* Raise it */
    counter1 = 0;
    while(counter1<100)
    {
      counter1++;
    }
/*Lower the level again .Lower GPIO25*/
   *(volatile uint16*)(0xc028)|= 0x0200;

}
```

app.c (in MSC_API subdirectory)

In the app_pre_init routine, the following was added to initialize the SPI hardware engine so SPI commands can be processed. This routine is not required when booted from the EEPROM but is still in the code to support memory loading from the SPI when an external EEPROM is not used. The code that was used to create *MSC_EEPROM_scan_LCP_v2.bin* has this code.

```
/* --- Initialize SPI interface by EWB  ----- */
   #ifdef EEPROM_Mode
      /* Typecast a function pointer */
         typedef void (*VOIDFUNC)(void);
      /* Create an instance of the VOIDFUNC and initialize */
      /* it to have the address you want to call into. */
         VOIDFUNC    my_spi_ginit = (VOIDFUNC) 0xe998;
      /* Make the call. */
         my_spi_ginit();

      /* Set SPI Enable in GPIO Control Reg */
      /*  *(volatile uint16*)(0xc006)|= 0x0020; */
      /*  *(volatile uint16*)(0xc03a)= 0x2222; */
      /*  *(volatile uint16*)(0xc038)= 0x0008; */

      /* __asm("sti"); */
   #endif /* EEPROM_Mode */
   /* ---- end EWB mods ---- */
```

Appendix C: Checksum Algorithm Implemented in C

```
/*************************************************************************
 * FUNCTION    : checksum
 * PURPOSE     : Calculate 1's complement checksum for the given block
**               of data in buffer specified by byte_length
 *
 * DESCRIPTION :
 * PARAMS      : Buffer Array, Length
 *
 * RETURNS     : 16 bit checksum
 *************************************************************************/


static uint16 checksum(char* buffer,uint16 byte_length)
{

 uint16 checksum;
 uint32 sum = 0;

        while( byte_length > 1 )
        {
           /*  This is the inner loop */
             sum += * (unsigned short*) buffer++;
       buffer++;

         byte_length -= 2;
        }

       /*  Add left-over byte, if any */
       if( byte_length > 0 )
           sum += * (unsigned char *) buffer;

       /*  Fold 32-bit sum to 16 bits */
       while (sum>>16)
           sum = (sum & 0xffff) + (sum >> 16);

       checksum = ~sum;

return(checksum);

}
#endif /*USE_CHECKSUM*/
```

Appendix D: SPI Initialization Sequence

After the Main Parameter Table address is known (e.g. 0x24E8), the pointers to the buffers and the location of the flags are known. Capture these values and then verify the Command Flag contains 0x55AA. Clear the Command Flag and the code is ready for the USB Flash drive. The clock rate in this design for the SPI bus is 1 MHz.

Index	min:sec.ms.us.ns	MOSI MISO	Comment
1	0:09.658.842.900	FFFF	Clear SPI Registers
2	0:09.658.910.800	FFFF	
3	0:09.658.978.700	FFFF	
4	0:09.659.046.700	FFFF	
5	0:09.659.114.600	FFFF	
6	0:09.659.182.500	FFFF	
7	0:09.659.250.400	FFFF	
8	0:09.659.318.400	FFFF	
9	0:09.659.390.200	04FF	Tell EZ-Host goto 04A4
10	0:09.659.461.900	CEFF	
11	0:09.659.533.500	A4FF	
12	0:09.659.605.100	04FF	
13	0:09.659.676.800	00FF	
14	0:09.659.748.400	00FF	
15	0:09.659.820.000	00FF	
16	0:09.659.891.700	00FF	<= GPIO25 pulses high after this byte. Wait for it.
17	0:10.006.775.200	10FF	Read Memory 0x0103 to Get Table Pointer
18	0:10.006.846.800	00FF	(Used to sync Status)
19	0:10.006.918.400	10FF	
20	0:10.006.990.100	03FF	
21	0:10.007.061.700	02FF	
22	0:10.007.133.300	00FF	
23	0:10.007.205.000	00FF	
24	0:10.007.276.600	00FF	
25	0:10.060.532.900	FFED	0x0FED from ‘goto’ command above
26	0:10.060.609.700	FF0F	
27	0:10.060.686.600	FF18	
28	0:10.060.763.400	FF00	
29	0:10.060.840.200	FF10	
30	0:10.060.917.100	FF03	
31	0:10.060.993.900	FF02	
32	0:10.061.070.700	FF00	
33	0:10.061.147.600	FF17	
34	0:10.061.224.400	FFDB	
35	0:10.061.301.200	FFE8	Contents of 0x0103
36	0:10.061.378.100	FF24	
37	0:10.114.641.500	18FF	
38	0:10.114.714.400	00FF	
39	0:10.114.787.400	10FF	
40	0:10.114.860.300	03FF	
41	0:10.114.933.300	02FF	
42	0:10.115.006.200	00FF	
43	0:10.115.079.200	17FF	
44	0:10.115.152.200	DBFF	
45	0:10.115.271.000	10FF	Read Memory 0x0103 to Get Table Pointer
46	0:10.115.343.900	00FF	
47	0:10.115.416.800	10FF	
48	0:10.115.489.700	03FF	
49	0:10.115.562.500	02FF	
50	0:10.115.635.400	00FF	
51	0:10.115.711.400	00FF	
52	0:10.115.784.300	00FF	
53	0:10.115.922.200	FF18	
54	0:10.115.995.200	FF00	
55	0:10.116.068.100	FF10	
56	0:10.116.141.100	FF03	
57	0:10.116.214.100	FF02	

```
58     0:10.116.287.000    FF00
59     0:10.116.358.000    FF17
60     0:10.116.430.900    FFDB
61     0:10.116.503.900    FFE8      Main Table is at 0x24E8
62     0:10.116.576.900    FF24
63     0:10.117.107.900    18FF
64     0:10.117.180.800    00FF
65     0:10.117.253.700    10FF
66     0:10.117.326.500    03FF
67     0:10.117.402.600    02FF
68     0:10.117.475.400    00FF
69     0:10.117.548.300    17FF
70     0:10.117.621.200    DBFF
```

Appendix E: SPI Example Sequence for Show Current Directory

This is an actual SPI bus sequence that shows the transactions for a SHOW CURRENT DIRECTORY function. The clock rate for the SPI bus is 1 MHz. Note the delays betweens each byte transferred. The slave select signal toggles for each byte. Also note the time between the different command segments (i.e. Write Memory command of eight bytes, the eight byte ACK, etc).

Index	min:sec.ms.us.ns	Data	Comment
0	0:00.000.000.000	MOSI MISO	
1	0:03.288.410.600	20FF	Write address command (8 bytes)
2	0:03.288.416.500	00FF	The Write Address is writing 6 bytes
3	0:03.288.422.500	FAFF	to address 0x24FA
4	0:03.288.428.500	24FF	
5	0:03.288.434.400	06FF	
6	0:03.288.440.400	00FF	
7	0:03.288.446.400	EAFF	
8	0:03.288.452.400	0EFF	
9	0:03.288.770.800	FF28	ACK for the Write address command (8 bytes)
10	0:03.288.776.800	FF00	
11	0:03.288.782.800	FFFA	
12	0:03.288.788.700	FF24	
13	0:03.288.794.700	FF06	
14	0:03.288.800.700	FF00	
15	0:03.288.806.600	FFEA	
16	0:03.288.812.600	FF0E	
17	0:03.289.195.100	0DFF	The 6 bytes are F00D, 0004, 0104 --- the CWD Operation
18	0:03.289.201.100	F0FF	
19	0:03.289.207.000	04FF	
20	0:03.289.213.000	00FF	
21	0:03.289.219.000	04FF	
22	0:03.289.224.900	01FF	
23	0:03.289.685.500	FF28	ACK again for the Write address command (8 bytes)
24	0:03.289.691.500	FF00	
25	0:03.289.697.500	FFFA	
26	0:03.289.703.400	FF24	
27	0:03.289.709.400	FF06	
28	0:03.289.715.400	FF00	
29	0:03.289.721.300	FFEA	
30	0:03.289.727.300	FF0E	
31	0:03.290.242.200	20FF	Now write the Command Flag 0x24F6…
32	0:03.290.248.100	00FF	
33	0:03.290.254.100	F6FF	
34	0:03.290.260.100	24FF	
35	0:03.290.266.000	02FF	
36	0:03.290.272.000	00FF	
37	0:03.290.278.000	49FF	
38	0:03.290.284.000	3CFF	
39	0:03.290.648.000	FF28	
40	0:03.290.654.500	FF00	
41	0:03.290.660.500	FFF6	
42	0:03.290.666.400	FF24	
43	0:03.290.672.400	FF02	
44	0:03.290.678.400	FF00	
45	0:03.290.684.300	FF49	
46	0:03.290.690.300	FF3C	
47	0:03.291.103.700	B6FF	with C3B6 --- tells EZ-Host to Execute CWD
48	0:03.291.109.700	C3FF	
49	0:03.291.578.400	FF28	
50	0:03.291.584.400	FF00	
51	0:03.291.590.400	FFF6	
52	0:03.291.596.300	FF24	
53	0:03.291.602.300	FF02	
54	0:03.291.608.300	FF00	
55	0:03.291.614.300	FF49	
56	0:03.291.620.200	FF3C	Wait for GPIO25 before proceeding after this byte
57	0:03.292.973.800	10FF	Now Read Main Table at 24E8

```
58     0:03.292.979.800    00FF
59     0:03.292.985.800    E8FF
60     0:03.292.991.700    24FF
61     0:03.292.997.700    14FF
62     0:03.293.003.700    00FF
63     0:03.293.009.700    00FF
64     0:03.293.015.600    00FF
65     0:03.293.381.300    FF18
66     0:03.293.387.300    FF00
67     0:03.293.393.200    FFE8
68     0:03.293.399.200    FF24
69     0:03.293.405.200    FF14
70     0:03.293.411.100    FF00
71     0:03.293.417.100    FF21
72     0:03.293.423.100    FF90
73     0:03.293.816.400    FF00   Status – 0000 is OK
74     0:03.293.822.400    FF00
75     0:03.293.828.300    FF00
76     0:03.293.834.300    FF00
77     0:03.293.840.300    FFFA
78     0:03.293.846.200    FF24
79     0:03.293.852.200    FF6A
80     0:03.293.858.200    FF25
81     0:03.293.864.100    FF6A
82     0:03.293.870.100    FF2D
83     0:03.293.876.100    FF00
84     0:03.293.882.100    FF08
85     0:03.293.888.000    FF02   Data Count – 2 bytes in Data Buffer
86     0:03.293.894.000    FF00
87     0:03.293.900.000    FF00
88     0:03.293.905.900    FF00
89     0:03.293.911.900    FF00
90     0:03.293.917.900    FF00
91     0:03.293.923.800    FF0D
92     0:03.293.929.800    FFF0
93     0:03.294.287.300    18FF
94     0:03.294.293.300    00FF
95     0:03.294.299.300    E8FF
96     0:03.294.305.200    24FF
97     0:03.294.311.200    14FF
98     0:03.294.317.200    00FF
99     0:03.294.323.100    21FF
100    0:03.294.329.100    90FF
101    0:03.294.844.500    10FF   Read the 2 bytes from Data Buffer (0x256A)
102    0:03.294.850.500    00FF
103    0:03.294.856.400    6AFF
104    0:03.294.862.400    25FF
105    0:03.294.868.400    02FF
106    0:03.294.874.300    00FF
107    0:03.294.880.300    00FF
108    0:03.294.886.300    00FF
109    0:03.295.272.600    FF18
110    0:03.295.278.500    FF00
111    0:03.295.284.500    FF6A
112    0:03.295.290.500    FF25
113    0:03.295.296.400    FF02
114    0:03.295.302.400    FF00
115    0:03.295.308.400    FFD0
116    0:03.295.314.300    FFFF
117    0:03.295.715.300    FF2F   This is the name of the directory, null terminated.
118    0:03.295.721.200    FF00   Since we're in the root, a slash (/) this there.
119    0:03.296.178.100    18FF
120    0:03.296.184.000    00FF
121    0:03.296.190.000    6AFF
122    0:03.296.196.000    25FF
```

```
123    0:03.296.201.900    02FF
124    0:03.296.207.900    00FF
125    0:03.296.213.900    D0FF
126    0:03.296.219.800    FFFF
127    0:03.296.753.700    10FF   Read Callback Flag (0x24F8)
128    0:03.296.759.700    00FF
129    0:03.296.765.600    F8FF
130    0:03.296.771.600    24FF
131    0:03.296.777.600    02FF
132    0:03.296.783.500    00FF
133    0:03.296.789.500    00FF
134    0:03.296.795.500    00FF
135    0:03.297.181.800    FF18
136    0:03.297.188.300    FF00
137    0:03.297.194.200    FFF8
138    0:03.297.200.200    FF24
139    0:03.297.206.200    FF02
140    0:03.297.212.100    FF00
141    0:03.297.218.100    FFFF
142    0:03.297.224.100    FFFF
143    0:03.297.619.000    FF00   Make sure 0000, otherwise USB may have been unplugged
144    0:03.297.625.000    FF00
145    0:03.298.090.000    18FF
146    0:03.298.095.900    00FF
147    0:03.298.101.900    F8FF
148    0:03.298.107.900    24FF
149    0:03.298.113.800    02FF
150    0:03.298.152.400    00FF
151    0:03.298.158.300    FFFF
152    0:03.298.164.300    FFFF
```

About the Author

Name: Ernie Buterbaugh

Title: Principal Field Applications Engineer

Background: BSEE from Pennsylvania State University. More than 25 years experience with embedded processors, board level design, ASIC, FPGA , and CPLD designs. Has authored a variety of application notes, articles, and the book Perfect Timing: A Design Guide for Clock Generation and Distribution.

Contact: ewb@cypress.com

Document History

Document Title: USB Flash Drive Controller Using SPI - AN15484

Document Number: 001-15484

Revision	ECN	Orig. of Change	Submission Date	Description of Change
**	1070141	NMMA	05/15/2007	New Application Note
*A	3056366	DBIR	10/12/2010	Updated title. Template Updated

Cypress Semiconductor
198 Champion Court
San Jose, CA 95134-1709
Phone: 408-943-2600
Fax: 408-943-4730
http://www.cypress.com/