PANDORA PRODUCTS

SD Card on LPC1114
SDCard

Jim Schimpf

Document Number: PAN-201612001
Revision Number: 0.9
6 January 2017

Pandora Products.
215 Uschak Road
Derry, PA 15627

Pandora Products.

215 Uschak Road

Derry, PA 15627

Phone: 724-539.1276

Email: jim.schimpf@gmail.com

Document Revision History

Version	Author	Description	Date
0.1	js	Initial Version	6-Dec-2016
0.2	js	More SPI speeds	9-Dec-2016
0.3	js	SD Card interface	9-Dec-2016
0.4	js	SD Card Initialization	10-Dec-2016
0.5	js	SD Block Reader	15-Dec-2016
0.6	js	SD Block Writer & FAT file system	25-Dec-2016
0.7	js	SDCard.c code	29-Dec-2016
0.8	js	Working & SDCard code complete	5-Jan-2017
0.9	js	Fix typos	6-Jan-2017

Contents

List of Figures

List of Tables

1 Introduction

An SD card adds a "disk drive" to embedded projects that allows them to store huge amounts of data and have a removable storage medium. I had some SD card socket assemblies and decided to try to interface them to the LPC1114. There are two parts to this interface, the hardware (SPI or SSP on NXP) and the software adding a FAT style file system. The hardware interface needs to be developed for the LPC1114 but there is already code for the FAT file system for SD cards which must be adapted for this processor.

2 SPI/SSP

2.1 Hardware

2.1.1 Processor Pins

The first item that must be done is to set up the pins on the processor for SPI0 use. The following pins are used.

Pin	Name	Use
1	PIO_8	MISO0
2	PIO_9	MOSI0
6	PIO_6	SCK0
25	PIO_2	SSEL0

Table 1: SPI pins

The SCK pin is a secondary choice because the other SCK0 is pin 3 which is already used for SWCLK a debug line. Set up these lines in the processor multiplexer section with following code.

```
#define LPC_SSP           LPC_SSP0
Chip_IOCON_PinMuxSet(LPC_IOCON, IOCON_PIO0_8, (IOCON_FUNC1 | IOCON_MODE_INACT));/* MISO0 */
Chip_IOCON_PinMuxSet(LPC_IOCON, IOCON_PIO0_9, (IOCON_FUNC1 | IOCON_MODE_INACT));/* MOSI0 */
Chip_IOCON_PinMuxSet(LPC_IOCON, IOCON_PIO0_2, (IOCON_FUNC1 | IOCON_MODE_INACT));/* SSEL0 */
Chip_IOCON_PinMuxSet(LPC_IOCON, IOCON_PIO0_6, (IOCON_FUNC2 | IOCON_MODE_INACT));/* SCK0 */
Chip_IOCON_PinLocSel(LPC_IOCON, IOCON_SCKLOC_PIO0_6);
Chip_SSP_Init(LPC_SSP);
```

Note that PIO0_6 needs to be set with IOCON_FUNC2 to enable it. All the lines are set MODE_INACT since they are under the control of the SPI block not the GPIO.

2.1.2 SPI Setup

Next the SPI must be set up for the clock phase, frame format and data bits. The following code does that.

```
// These are in the ssp_11xx.h file
typedef enum CHIP_SSP_FRAME_FORMAT {
SSP_FRAMEFORMAT_SPI = (0 << 4),
/**< Frame format: SPI */
CHIP_SSP_FRAME_FORMAT_TI = (1u << 4),/**< Frame format: TI SSI */
SSP_FRAMEFORMAT_MICROWIRE = (2u << 4),/**< Frame format: Microwire */
} CHIP_SSP_FRAME_FORMAT_T;
typedef enum CHIP_SSP_CLOCK_FORMAT {
SSP_CLOCK_CPHA0_CPOL0 = (0 << 6),/**< CPHA = 0, CPOL = 0 */
SSP_CLOCK_CPHA0_CPOL1 = (1u << 6),/**< CPHA = 0, CPOL = 1 */
SSP_CLOCK_CPHA1_CPOL0 = (2u << 6),/**< CPHA = 1, CPOL = 0 */
SSP_CLOCK_CPHA1_CPOL1 = (3u << 6),/**< CPHA = 1, CPOL = 1 */
SSP_CLOCK_MODE0 = SSP_CLOCK_CPHA0_CPOL0,/**< alias */
SSP_CLOCK_MODE1 = SSP_CLOCK_CPHA1_CPOL0,/**< alias */
SSP_CLOCK_MODE2 = SSP_CLOCK_CPHA0_CPOL1,/**< alias */
SSP_CLOCK_MODE3 = SSP_CLOCK_CPHA1_CPOL1,/**< alias */
} CHIP_SSP_CLOCK_MODE_T;
// ********************************
#define SSP_MODE_TEST       1 /*1: Master, 0: Slave */
SSP_ConfigFormat ssp_format;
ssp_format.frameFormat = SSP_FRAMEFORMAT_SPI;
ssp_format.bits = SSP_DATA_BITS;
ssp_format.clockMode = SSP_CLOCK_MODE0;
Chip_SSP_SetFormat(LPC_SSP, ssp_format.bits,
ssp_format.frameFormat,
ssp_format.clockMode);
Chip_SSP_SetMaster(LPC_SSP, SSP_MODE_TEST);
Chip_SSP_Enable(LPC_SSP);
Chip_SSP_DisableLoopBack(LPC_SSP);
```

Last the clock speed is set with a call to Chi0_SSP_SetClockRate(); The call has the following parameters:

```
/**
* @brief Set up output clocks per bit for SSP bus
* @param pSSP : The base of SSP peripheral on the chip
* @param clk_rate fs: The number of prescaler-output clocks per bit on the bus, minus one
* @param prescale : The factor by which the Prescaler divides the SSP peripheral clock PCLK
* @return Nothing
* @note The bit frequency is PCLK / (prescale x[clk_rate+1])
*/
```

The output clocks for a number of input values are.

clk_rate	prescale	SCK mHz
1	2	11.99
1	4	6.099
1	8	3.012
1	16	1.507
1	32	754.0 (kHz)
2	2	8.054
2	4	4.032
2	8	2.012
2	16	1.007
2	32	502.1 (kHz)
3	2	6.030
3	4	3.017
3	8	1.509
3	16	753.5 (kHz)
3	32	376.8 (kHz)

Table 2: Clock speeds

This is with the default 48 mHz clock speed and using the built in oscillator.

2.2 Software

2.2.1 API

The following API has been written for SPI use:

2.2.2 void SPI_Pins(void) - Initialize SPI I/O pins

Input	Use	Output
	Set up SPI pins	
		NONE

This call sets up the spi pins a shown in Figure 1 on page 2.

2.2.3 void SPI_config(SSP_ConfigFormat *fmt,uint32_t rate,uint32_t pre) Set up configuration

Input	Use	Output
fmt	Bits & format	
rate	Clock rate/bit -1	
pre	Prescale value	
		NONE

This sets up the SPI configuration the fmt is an SSP_ConfigFormat object with values for bits/word, frameFormat and clockMode. The rate and pre values set the SPI clock shown in Figure 2 on the previous page. The values for the fmt fields are shown here in section 2.1.2 on page 2.

2.2.4 int SPI_xfer(int size,unsigned char *in, unsigned char *out) - Transfer SPI data

Input	Use	Output
size	# Bytes to transfer	
in	Data sent to SPI	
out	Data returned from SPI	
		# bytes in xfer

This is the call to actually write data to/from the SPI device. The flag POLLING_MODE determines if the interrupt or polling method is used. If POLLING_MODE = 1 then the polling call is made and the SPI_xfer is synchronous and on return the return value is the number of bytes returned from the read.

If POLLING_MODE = 0 and INTERRUPT_MODE is used then the call to SPI_xfer is asynchronous and returns immediately with the return value = to size. See SPI_Done() 2.2.5

2.2.5 int SPI_Done(void) - Check for xfer done

Input	Use	Output
		=0 not done, # bytes transferred if don

If INTERRUPT_MODE = 1, i.e. interrupt mode is used then this call returns 0 till the transfer is done and then returns the # of bytes transferred when the transfer is complete.

2.3 Software Use

As noted above the API has polling and interrupt modes. The difference occurs when you call SPI_xfer() in polling mode this call is blocking and in interrupt mode the call is non-blocking and SPI_Done() tells when it is finished. There is very little speed difference in the transfer of 256 bytes in polling or interrupt mode.

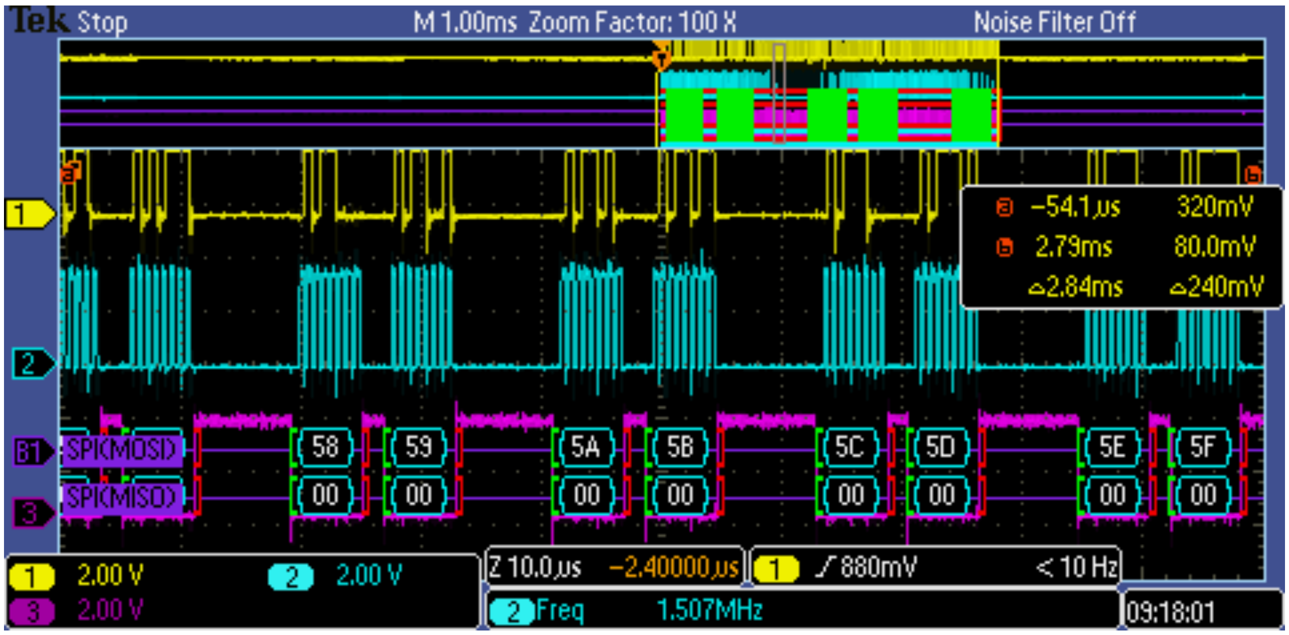

Figure 1: 256 bytes xfer in Polling Mode

as you can see it took about 2.84 ms to transfer.

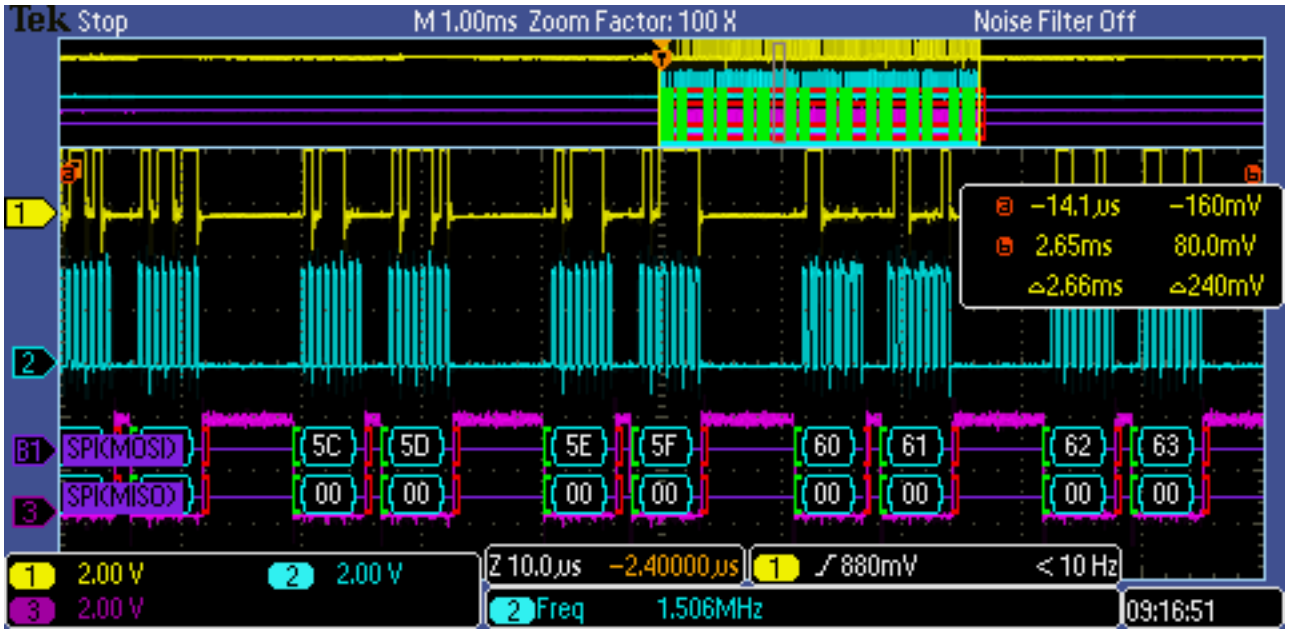

Figure 2: 256 bytes xfer in Interrupt Mode

The interrupt was a little faster at 2.66 ms for the same transfer.

Also the clock speed of SCK does not seem to matter both the above transfers were done at 1.5 MHz. Here is a transfer done at 12 MHz.

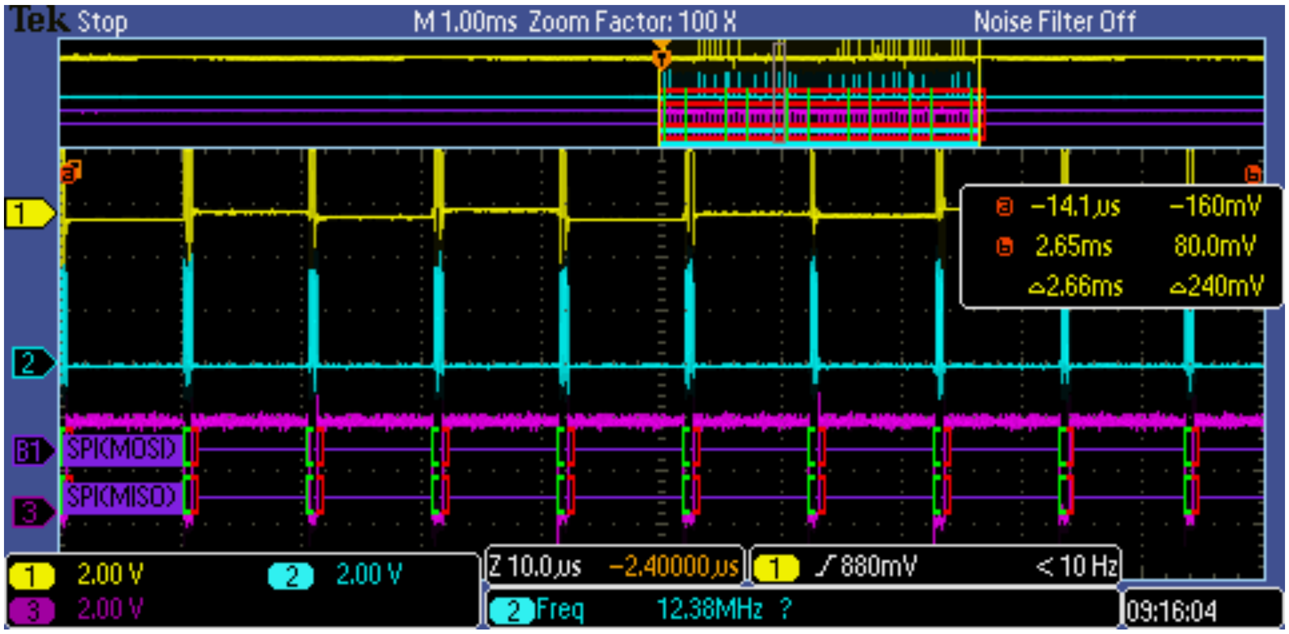

Figure 3: 12 MHz 256 byte xfer in Interrupt Mode

As you can see it takes the same time but the time/byte is smaller but the gaps between sending bytes is still there.

3 SD Card

3.1 Introduction

After the SPI/SSP is working we must then interface to a Secure Digital card (SD Card) and then we have to media to build a file system. I found this paper [3] at this location http://alumni.cs.ucr.edu/~amitra/sdcard/Additional/sdcard_appnote_foust.pdf that describes how to interface to an SDCard. The software in this section will be based on this paper.

3.2 Hardware

3.2.1 Socket

The hardware used was 1PCS SD Card Socket Module Slot Reader For Arduino ARM MCU LW from eBay [2] http://www.ebay.com/itm/1PCS-SD-Card-Socket-Module-Slot-Reader-For-Arc
172144420638?hash=item28149b671e:g:TigAAOSwXeJXdj51

This unit takes a full size SD card and has the SPI lines brought out to a header It also has a 5V -> 3.3V power supply chip on it but in this design the 3.3V only is used. It also supplies all the pull-up resistors shown in [3] in Figure 2 page 3. It does NOT have an SD card inserted switch.

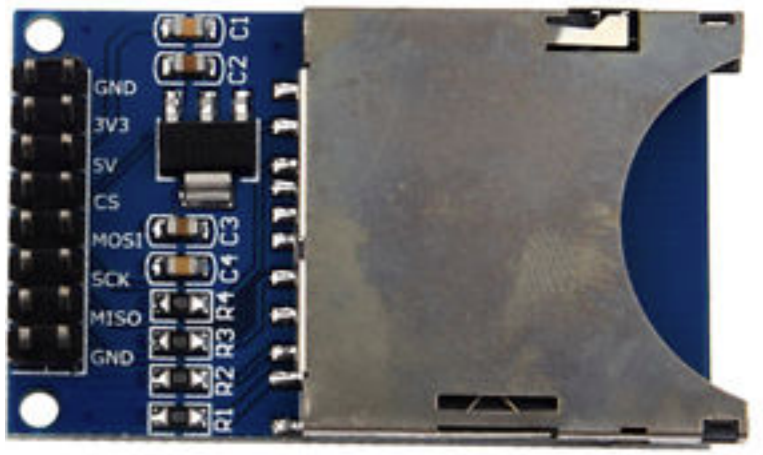

Figure 4: SD Card Module

3.2.2 CS Signal

The CS signal is NOT the SSEL signal from SPI, that signal goes low on every byte transmitted then high between bytes. For the SD card the SS signal has to stay low over a number of SPI transfers. Instead PIO0_7 is enabled as a GPIO and the routine SS_CC(flag) is used to control it explicitly. CS is set low in the initialization during the entire sequence.

3.3 SD Software low level

3.3.1 Sending Commands

All the code following uses the routine SD_SendCmd() which sends a command to the card and returns the status. When a command is sent it is a 6 byte block with the command as the first byte, a 4 byte big-endian argument then one byte of a CRC-7 of the proceeding 5 bytes. The routine builds this structure with the argument passed in as 32 bit unsigned integer.

Each command returns 1,2 or 5 bytes of response. The first byte is a status byte and the response is not valid till the SD card returns a byte with the top bit clear. The routine reads SPI data till this is found and then reads the rest (if any of the response). Also the routine sticks in 2 extra bytes as padding to give the SD card time to handle the command. Also the card is selected when the command is sent and the response received and left selected on exit of the routine.

3.3.2 Initialization

Before a card can be used it must be initialized. As explained in [3] there are 3 interfaces to the card and we are going to bring up the SPI interface. The SPI initialization is shown on page 7 of [3]. The code in SD_Init() follows this chart and the code on page 10. The clock speed of SPI is set to 376.6 KHz, which is the closest we could get (see Table 2 on page 4) to 400 KHz.

The initialization code was copied from code found `https://developer.mbed.org/compiler/#nav:/HelloWorld/SDFileSystem;` here. Looking at the code in Hello World project, SD-FileSystem.cpp. First 80 clocks (i.e. 10 bytes of 0xFF) are transmitted to synchronize the card. Then CMD0 is sent to initialize the card. Next CMD8 is sent with an argument of 0x1AA. On a type 1 card this returns a status with the Illegal command bit clear, on a type 2 card (most modern cards) this returns with the Illegal Command bit set. Once this is done then a loop is entered sending a CMD55 (prep for an ACMD command) then ACMD41 and keep sending this pattern till the status byte returned has bit 0 (Idle bit) == 0. This takes a long time, there is a 55 ms delay is this loop and it takes a couple of cycles before the card is ready.

Finally CMD58 is read to get the OCR and check bit there. Then CMD16 is sent to set the block size for 512 bytes and the card is ready for use. Also on successful initialization the SPI clock is set to 1.509 MHz from the slower initialization speed.

3.3.3 SD_Init

int SD_Init(void)

Input	Use	Output
	Result	0 => success, 0< => failures

3.3.4 Block Read

The block read is relatively simple, CMD17 is sent with the argument set to the block desired. Then a loop is entered with the card selected waiting for the card to send back the token 0xFE. This takes on the order of 5ms before this is received. Once received the block (512 bytes) are read via SPI. The image shows the 0xFE being received and the data transfer starting. The upper traces show the whole transaction, with the CMD17 at the left, the wait for 0xFE and the data transfer

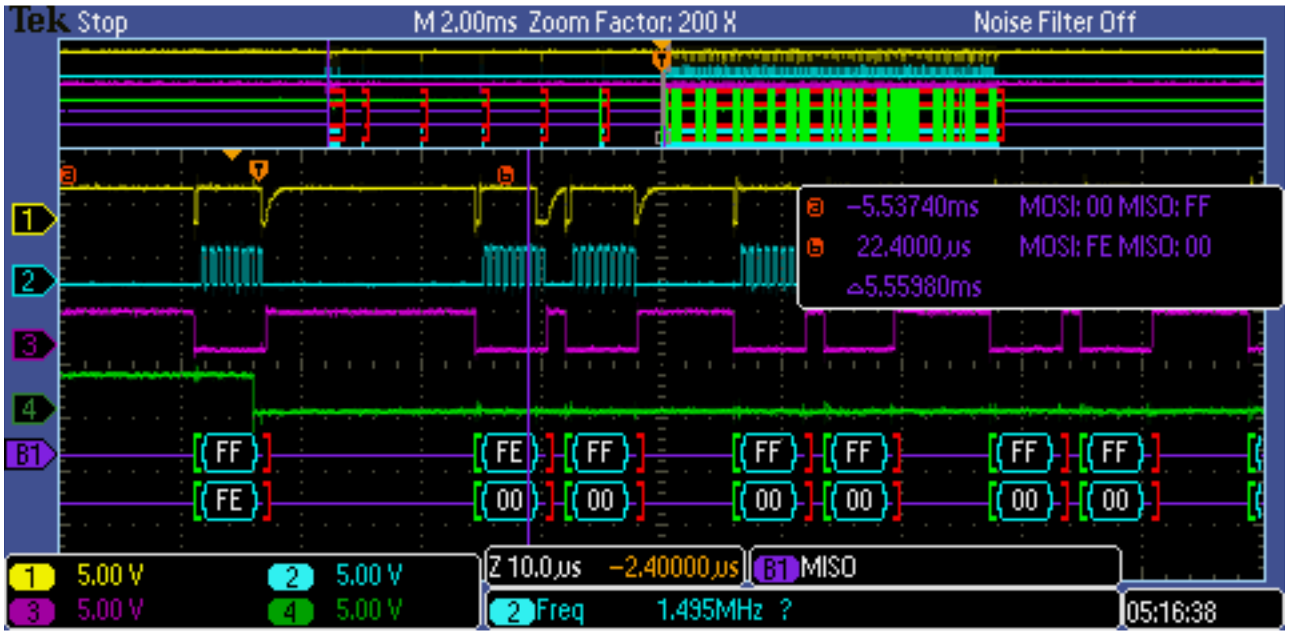

Figure 5: Read Operation

3.3.4.1 SD_ReadBlock

```
int SD_ReadBlock (uint32_t blockaddr,uint8_t *data)
```

Input	Use	Output
blockaddr	Absolute block # on SD card	
data	Data buffer of 512 bytes	Block read from card
	Result	0 => success 0< implies failure

3.3.5 Block Write

Single block write is quite simple. Issue the command (CMD24) and get a response with the bottom bit cleared. Send a start of data mark 0xFE. Once that is done send the data block and then following the data block another 16 bits of data (0xFFFF) to simulate the CRC which is ignored. After this the card select line is disabled.

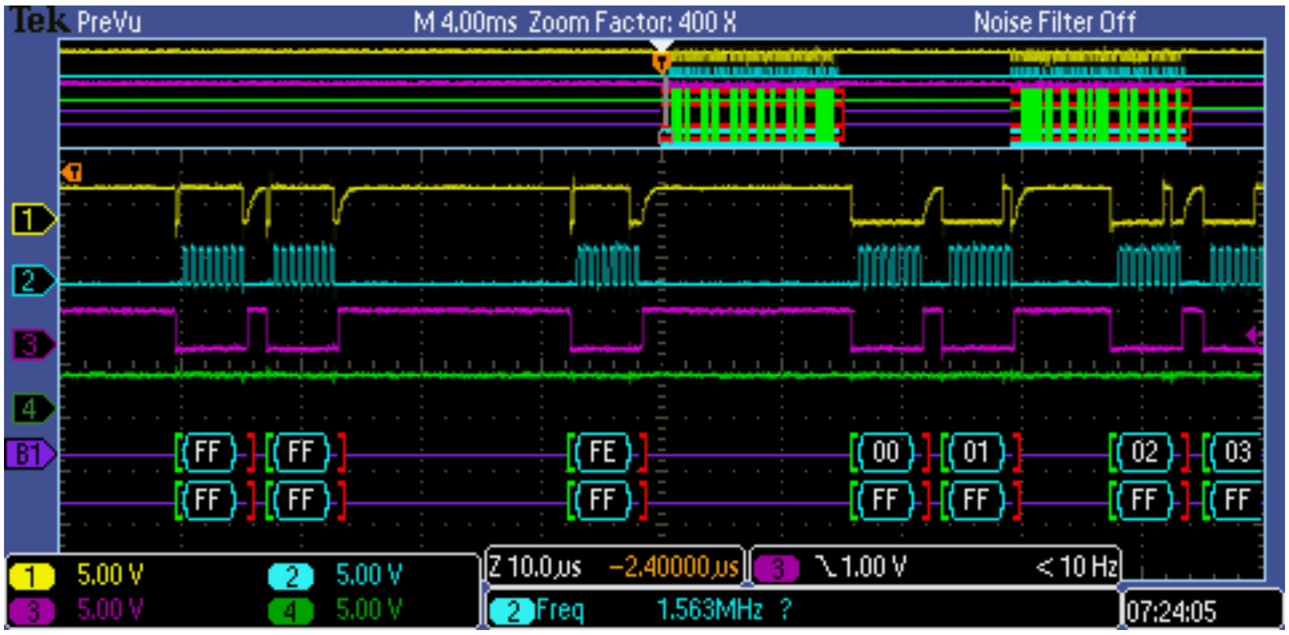

Figure 6: Write Data

This shows the start of data mark and then the data (which was 1,2,3... 255). The block write was repeated twice as shown by the upper traces.

3.3.5.1 SD_WriteBlock

```
int SD_WriteBlock (uint32_t blockaddr,uint8_t *data)
```

Input	Use	Output
blockaddr	Absolute block # on SD card	
data	Data buffer of 512 bytes	Written to card
	Result	0 => success 0< implies failure

3.4 FAT File System[4]

The structure and operation of the FAT file system is explained here https://en.wikipedia.org/wiki/File_Allocation_Table. It is the most common file system used in SD cards in a variety of devices. There are very good embedded versions of the system available (see below). See the article above for more information on the internals of the system.

With the initialize card, read a block and write a block working the FAT (File Allocation Block) file system can be supported. See [1] at http://elm-chan.org/fsw/ff/00index_e.html for excellent documentation and source code for an embedded version of this file system.

The code has an interface section called diskio.c that has the disk operation routines called by the FAT code and in them you put the calls to previously mentioned initialize, read and write routines. Then the rest of the FAT code "just works". The above mentioned URL also has links to the documentation on the FAT API and configuration of the whole software system. The configuration allow customization of how much of the API is present and also for thing lacking in embedded systems like a real time clock.

3.4.1 diskio Modifications

The diskio code is designed to handle RAM,SD and USB devices. In this application only the SD card is the device in use so the code is modified to handle just it. The code is changed so that only the SD device section is active in each of the routines in the module.

3.4.1.1 disk_initialize This section is called to setup the SD card for use. In addition a flag variable **inited** is added to the routine as a static to mark that the SD card has been successfully initialized. The routine can be called multiple times but will only initialize the SD card once. This routine called the SD_Init() 3.3.3 on page 9 from the SD card code to do the actual work. It's success or failure is recorded in the **inited** flag.

3.4.1.2 disk_initialize This routine is quite simple, it returns the state of the **inited** flag. It is called before other actions are taken to check if the code needs to initialize the SD card if it has not been done before.

3.4.1.3 disk_read and disk_write These two routines are simular, both use the single block read and write (3.3.4.1 on page 10 and 3.3.5.1 on the previous page). The count value passed is the number of blocks to read or write and a loop calling the read or write single block routine is used to handle this. Also a re-try mechanism is added to re-try an operation if a failure is returned. If if fails after 3 tries then the routine returns a failure.

3.5 Main Program

3.5.1 Introduction

The main program (SDCard.c) ties all the pieces together and allows the parts to be tested. It is a simple menu based system allowing you to do the low level functions and use the FAT file functions. The whole system runs under FreeRTOS and the menu system actually runs in a thread.

```
SD CARD Control Program VER:[Jan  5 2017 05:08:09]
SD Card Control
      -- Card --
1 - Init Card
```

```
2 - Read Card Sec
3 - Write Card Sec
     -- FAT FILE --
4 - Mount Card
5 - Directory
6 - Read File
7 - Write File
8 - Delete File
Choice:
```

3.5.2 Card Functions

These call the the card level functions see 3.3 on page 8. It was orignally written to allow easy testing of these functions. Each call the functions laid out in that section. The read card also dumps the read data out to the terminal so you can look at it.

3.5.3 FAT Functions

These call the FAT file system API and allow it to be checked. The **Mount Card** initializes the card for use. This was done to test the **disk_initialize** call in the diskio.c code explicitly. The other functions just exercise the FAT API and are more a check on the diskio.c functions that the FAT code.

3.5.3.1 Mount Card This mounts the FAT file system and initializes the card. This operation will automatically be called in the other FAT functions if not called here. Also it set a flag so it only gets called once.

3.5.3.2 Directory This is a simple directory read one level deep that reads all the files at the top level on the SD card and sums their sizes.

3.5.3.3 Read File Input a file name and the text of the file is read and dumped to the serial port.

3.5.3.4 Write File This writes a test pattern "0123456789ABCDEF " to a file name specified by the user. It can create files or overwrite an existing file. The test pattern is written 100 times to the file.

3.5.3.5 Delete File This deletes a specified file.

3.5.4 Getting it working

These FAT functions kept failing and it was found that the fundamental disk read and write functions were failing. It was found that the basic functions would work successfully after SD_Init() was called and then fail when called subsuquently. So a call to SD_Init() was added to sector read and write calls. This slows them down "a lot..." but does make the FAT system work successfully. Futher work will be done to determine what "reset" functions are actaually needed to allow these to work successfully.

References

[1] ChanN. Fatfs - generic fat file system module.

[2] e Bay. 1pcs sd card socket module slot reader for arduino arm mcu lw.

[3] C Fouts. Secure digital card interface for the msp430.

[4] Wikipeida.org. File allocation table.