

**A
PROJECT REPORT ON**

Linux USB Device Driver for File Transfer

**SUBMITTED IN
PARTIAL FULFILLMENT OF
DIPLOMA IN EMBEDDED SYSTEM DESIGN (PG-DESD)**



**BY
CHEBROLU LOKESH**

**AT
SUNBEAM INSTITUTE OF INFORMATION TECHNOLOGY,
PUNE**

**SUNBEAM INSTITUTE OF INFORMATION TECHNOLOGY,
PUNE.**



CERTIFICATE

This is to certify that the project

Linux USB Device Driver for File Transfer

Has been submitted by

CHEBROLU LOKESH

In partial fulfillment of the requirement for the Course of **PG Diploma in Embedded System Design (PG-DESD AUG2017)** as prescribed by The **CDAC ACTS, PUNE.**

Place: Pune

Date: 30-JAN-2018

Authorized Signature

ACKNOWLEDGEMENT

It is our privilege to acknowledge with deep sense of gratitude to our project mentor Sir Devendra Dhande for his valuable suggestions and guidance throughout our course of study and project. We are thankful to Sir Nilesh Ghule for his valuable guidance. We are highly obliged to the entire staff of DESD for their kind co-operation and help. We also take this opportunity to thank all our colleagues who backed our interest by giving useful suggestion and all possible help.

ABSTRACT

This project 'Linux Device Driver for File Transfer' consists of 3 parts which are Device, Device Driver and Test program. Device implements a firmware of LPC1768 blueboard, USB stack of Bertrik Sikken and ultra-FAT IO library was ported on ARM CortexM3 LPC1768. The files are kept on SD card connected to blueboard. User program lists files from the device and then fetch a file or can send a file to device.

INDEX

1.	INTRODUCTION	7
	1.1 Introduction	7
	1.2 Overview of Project	7
2.	LITERATURE SURVEY	8
	USB	8
	SSP Protocol	11
	FAT File System	13
3.	PROJECT ARCHITECTURE	16
	Device	16
	Host PC	17
4.	HARDWARE DESCRIPTION	19
	LPC1768	19
	USB	20
	SSP	23
	SD Card	24
5.	SOFTWARE REQUIREMENTS AND SPECIFICATIONS	25
6.	SOURCE CODE EXPLANATION	27
7.	TESTING	34
8.	FUTURE SCOPE	38
9.	CONCLUSION	39
10.	REFERNCES	40

LIST OF TABLES

S. No	Table Title	Page
1	FAT file systems differences	14
2	FAT Partition Boot Sector	15
3	USB Pin Configuration	21
4	USB Clock	21

LIST OF FIGURES

S. No	Figure Title	Page
1	Simplest SPI Configuration	12
2	SPI Master Output clock signal	12
3	SPI Master activating slave	12
4	SPI Master sending data	13
5	SPI Master reads data from slave	13
6	Project Architecture	16
7	LPC1768 - Blueboard	19
8	SD Card Pin Description	24
9	Driver Module Insertion	34
10	Driver file creation	34
11	Test program execution	35
12	Write a file to SD Card	35
13	Reading file from SD Card	35
14	List file operation	36
15	Copy file from SD Card	36
16	Total dmesgs of process	37

1.Introduction

1.1 Introduction:

Data acquisition in embedded systems became a crucial part now a days. That data can be like sensor data which need to be logged daily and need to be stored. For storing SD card needed to be used. USB became so universal that almost all appliances are supporting USB now. So considering these requirement this project is developed which includes file storing, reading into SD Card on LPC1768 and those files are read to a PC through USB which is being done using a Host Linux USB Device Driver on PC and User test program.

1.2 Overview of Project:

This project is divided into 3 parts which are Device, Device Driver and Test program. Device includes a LPC1768 Blueboard interfaced with SD Card which has FAT32 file system in it. Device driver includes a Linux USB device driver for device which is loaded into a host PC. Test program includes a user level application from which read, write operations on device can be performed through the driver which is loaded.

2. LITERATURE SURVEY

1. Introduction

USB is an interface that connects a device to a computer. With this connection the computer sends or retrieves data from the device. USB gives developers a standard interface to use in many different types of applications. A USB device is easy to connect and use because of a systematic design process.

2. History

USB is an industry standard developed for the connection of electronic peripherals such as keyboard, mice, modems, and hard drives to a computer.

This standard was developed in order to replace larger and slower connections such as serial and parallel ports. The standard was developed through a joint effort, starting in 1994, between Compaq, DEC, IBM, Intel, Microsoft, NEC, and Nortel. The goals were to develop a single interface that could be used across multiple devices, eliminate the many different connectors currently available at the time, and increase the data throughput of electronic devices.

3. How is data sent across the USB?

When the software requires data transfer to occur between itself and the USB, it sends a block of data called an I/O Request Packet (IRP) to the appropriate pipe, and the software is later notified when this request is completed successfully or terminated by error. Other than the presence of an IRP request, the pipe has no interaction with the USB. In the event of an error after three retry attempts, the IRP is cancelled and all further and outstanding IRPs to that pipe are ignored until the software responds to the error signal that is generated by sending an appropriate call to the USB. How exactly this is handled depends upon the type of device and the software.

4. Types of data transfer

- **Control Transfer:-**These differ from the other types in that they are intended for use in configuring, controlling, and checking the status of a USB device.

- Isochronous transfer: These involve data whose accuracy is not critical and which is sent at a rate corresponding to some timing mechanism.
- Interrupt transfers: These are used for small, infrequent transfers which require priority over other requests.
- Bulk transfer: As the name suggests, the intended purpose is for transmitting large amounts of data.

5. USB Enumeration and Configuration

1. Step 1: The device is connected to a USB port and detected. At this point, the device can draw up to 100 mA from the bus. The device is currently in the powered state.
2. Step 2: The hub detects the device by monitoring voltages on the ports. A hub has pull-down resistors on the D+ and D- lines. As mentioned earlier, there is a pull-up resistor on either the D+ or D- line depending on device speed. By monitoring the voltage transition on these lines, the hub detects if a device is attached.
3. Step 3: The host learns of the newly attached device by using an interrupt endpoint to get a report about the hub's status. This includes changes in port status. After the hub tells the host about the device detection, the host issues a request to the hub to learn more details about the status change that occurred using the GET_PORT_STATUS request.
4. Step 4: After the host gathers this information, it detects the speed of the device using the method mentioned in the USB Speeds. Initially, only Full-Speed or Low-Speed is detected by the hub by detecting if the pull-up resistor is on the D+ or D-line. This information is then reported to the host by another GET_PORT_STATUS request.
5. Step 5: The host issues a SET_PORT_FEATURE request to the hub asking it to reset the newly attached device. The device is put into a reset state by pulling both the D+ and D- lines down to GND (0 V). Holding these lines low for greater than 2.5 μ s issues the reset condition. This reset state is held for 10 ms by the hub.
6. Step 6: During this reset, a series of J-State and K-State occurs to determine if the device supports High-Speed. During this reset state, if the device supports High-Speed, it issues a single K-State. A High-Speed hub detects this K-State and responds with a sequence of J and K states to form a —KJKJKJ pattern. The device detects this pattern and removes its pull-up resistor from its D+ line. This step is skipped on Low-Speed and Full-Speed devices.

7. Step 7: The host then checks to see if the device is still in a reset state by issuing a GET_PORT_STATUS request. If the request reports that the device is still in reset, then the host continues to issue the request until it receives word that the device is out of reset. After the device leaves reset, it is in the default state as mentioned in the USB Power section. The device can now respond to requests from the host in the form of control transfers to its default address of 00h. All USB devices start with this default address. Only one USB device can have this address at a time; this is why when you connect multiple USB devices to a port at the same time, they enumerate sequentially and not simultaneously.
8. Step 8: The host begins the process of learning more information about the device. It starts by learning the maximum packet size of the default pipe (Endpoint 0). The host starts by issuing a GET_DESCRIPTOR request to the device. In the device descriptor, the eighth byte (bMaxPacketSize0) contains information about the maximum packet size for EP0. A Windows host requests 64-bytes, but after only receiving 8 bytes of the device descriptor, it moves onto the status stage of the control transfer and requests that the hub reset the device. The USB specification requires that a device return at least 8 bytes of the device descriptor, when requested, if the device has the default address of 00h. The reason for requesting the 64-bytes is to avoid unpredictable behavior from the device. Additionally, the reason for performing a reset after only receiving 8 bytes is an artifact of early USB devices. In the early life of USB, some devices did not respond properly when a second request for the device descriptor occurred. To solve this problem, a reset after the first device descriptor request was required. Regardless, the 8 bytes that were transferred was enough to get the require information about the bMaxPacketSize0.
9. Step 9: The host applies an address to the device with the SET_ADDRESS request. The device completes the status stage of this request using the default 00h address before using the newly assigned address. All communication beyond this point will use the new address. The address may change if the device is detached from a port, the port is reset, or the PC reboots. The device is now in the address state.
10. Step 10: After the device returns from its reset, the host issues a command, GET_DESCRIPTOR, using the newly assigned address, to read the descriptors from the device. However, this time all the descriptors are read. The host uses this information to learn about the device and its abilities. This information includes the number of peripheral interfaces, power connection method, and the required maximum power. The host starts by requesting the device descriptor, but this time it receives the entire descriptor and not just a partial version. Next the host issues another GET_DESCRIPTOR command asking for the configuration descriptor. This request not only returns the configuration descriptor, but all other descriptors

associated with it such as the interface descriptor and the endpoint descriptor. A Windows PC first asks for just the configuration descriptor (9 bytes), then it issued a second GET_DESCRIPTOR request for the configuration descriptor and all other associated descriptors with that configuration (interface and endpoint descriptors).

11. Step 11: For the host PC to successfully use the device, a Windows PC in this case, the host must load a device driver. The host searches for a driver to manage communication between itself and the device. Windows uses its .inf files to locate a match for the devices Product ID and Vendor ID. Device release version numbers can optionally be used. If Windows cannot find a match, then it looks at the driver from a different perspective by looking for a match with any class, subclass, and protocol retrieved from the device. If a device was previously enumerated, Windows uses its registry to search for the proper driver. When a driver is identified, the host may request additional descriptors that are specific to the device class or request that descriptors are resent.
12. Step 12: After all descriptors are received, the host sets a specific device configuration using the SET_CONFIGURATION request. Most devices have only one configuration. Devices that support multiple configurations can allow the user or the driver to select the proper configuration.
13. Step 13: The device is now in the configured state. It took on its proprieties that were defined with the descriptors. The defined maximum power can be drawn from V BUS and the device is now ready for use in an application.

SPI PROTOCOL:

1. Introduction

Serial Peripheral Interface (SPI) is an interface bus commonly used to send data between microcontrollers and small peripherals such as shift registers, sensors, and SD cards. It uses separate clock and data lines, along with a select line to choose the device you wish to talk to. One unique benefit of

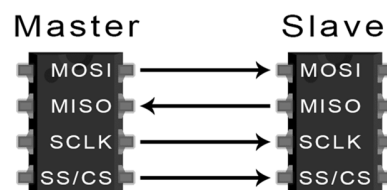


Figure 1: Simplest SPI Configuration

MOSI (Master Output/Slave Input) – Line for the master to send data to the slave.

MISO (Master Input/Slave Output) – Line for the slave to send data to the master.

SCLK (Clock) – Line for the clock signal.

SS/CS (Slave Select/Chip Select) – Line for the master to select which slave to send data to.

Steps for SPI data transmission:

- The master outputs the clock signal

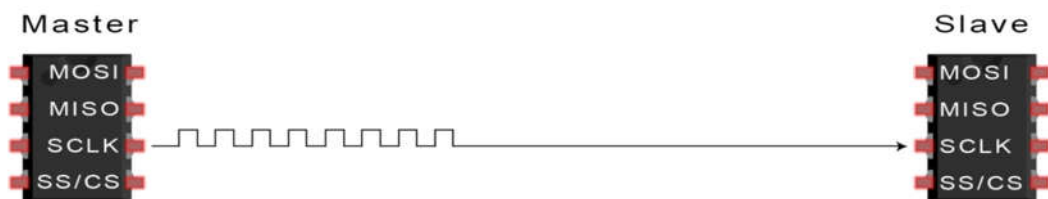


Figure 2: SPI Master Output the clock signal

- The master switches the SS/CS pin to a low voltage state, which activates the slave.

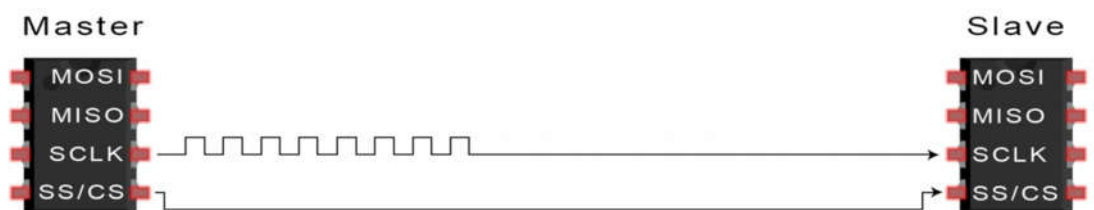


Figure 3: SPI Master activating slave

- The master sends the data one bit at a time to the slave along the MOSI line. The slave reads the bits as they are received:

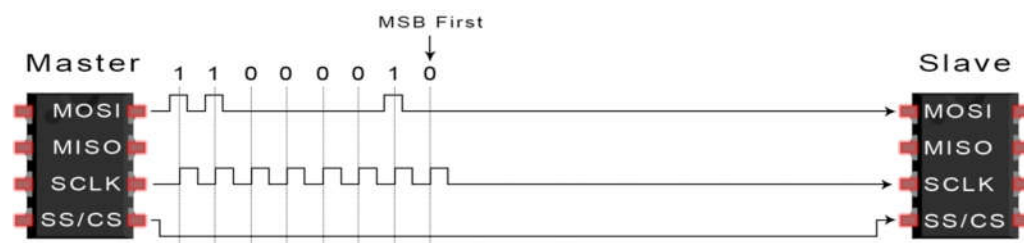


Figure 4: SPI Master sending data

- If a response is needed, the slave returns data one bit at a time to the master along the MISO line. The master reads the bits as they are received:

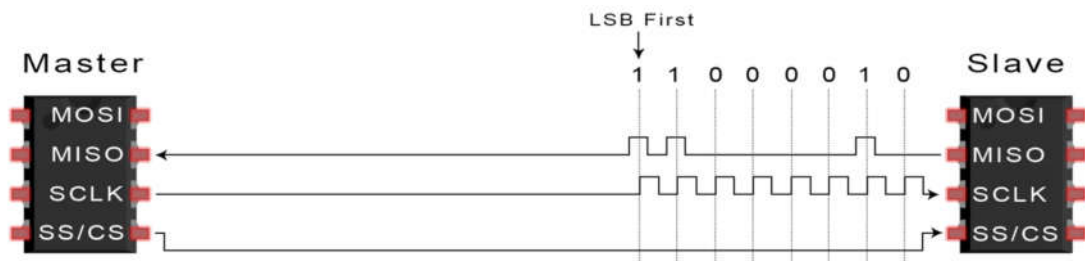


Figure 5: SPI Master reads data from slave

Advantages:

- No start and stop bits, so the data can be streamed continuously without interruption
- No complicated slave addressing system like I2C
- Higher data transfer rate than I2C (almost twice as fast)
- Separate MISO and MOSI lines, so data can be sent and received at the same time

Disadvantage:

- Uses four wires (I2C and UARTs use two)
- No acknowledgement that the data has been successfully received (I2C has this)
- No form of error checking like the parity bit in UART
- Only allows for a single master

FAT file system:

The File Allocation Table (FAT) file system is a simple file system originally designed for small disks and simple folder structures. The FAT file system is named for its method of organization, the file allocation table, which resides at the beginning of the volume. To protect the volume, two copies of the table are kept, in case one becomes damaged. In addition, the file allocation tables and the root folder must be stored in a fixed location so that the files needed to start the system can be correctly located.

A volume formatted with the FAT file system is allocated in clusters. The default cluster size is determined by the size of the volume. For the FAT file system, the cluster number must fit in 16 bits and must be a power of two.

Structure of a FAT volume

Partition Boot Sector	FAT1	FAT2 (duplicate)	Root Folder	Other folders and allocated files
--------------------------	------	---------------------	-------------	--------------------------------------

Differences between the FAT systems

System	Bytes Per Cluster Within FAT	Cluster Limit
FAT12	1.5	Fewer than 4087 clusters
FAT16	2	Between 4087 and 65526 clusters, inclusive
FAT32	4	Between 65526 and 268,435,456 clusters, inclusive

Table 1: FAT file systems differences

FAT32

- **File System Specifications:** FAT32 is a derivative of the File Allocation Table (FAT) file system that supports drives with over 2GB of storage. Because FAT32 drives can contain more than 65,526 clusters, smaller clusters are used than on large FAT16 drives. This method results in more efficient space allocation on the FAT32 drive. The largest possible file for a FAT32 drive is 4GB minus 2 bytes.
- The FAT32 file system includes four bytes per cluster within the file allocation table. Note that the high 4 bits of the 32-bit values in the FAT32 file allocation table are reserved and are not part of the cluster number

Byte Offset (in hex)	Field Length	Sample Value	Meaning
00	3 bytes	EB 3C 90	Jump instruction
03	8 bytes	MSDOS5.0	OEM Name in text
0B	25 bytes		BIOS Parameter Block
24	26 bytes		Extended BIOS Parameter Block
3E	448 bytes		Bootstrap code
1FE	2 bytes	0x55AA	End of sector marker

Table 2: FAT Partition Boot Sector

1. **BIOS Parameter Block:** The BPB for FAT32 drives is an extended version of the FAT16/FAT12 BPB. It contains identical information to a standard BPB, but also includes several extra fields for FAT32 specific information. This structure is implemented in Windows OEM Service Release 2 and later.

2. BIG FAT BOOT FS INFO Members:

Member Name	Description
bFSInf_Sig	The signature of the file system information sector. The value in this member is FSINFOSIG (0x61417272L).
bFSInf_free_clus_cnt	The count of free clusters on the drive. Set to -1 when the count is unknown.
bFSInf_next_free_clus	The cluster number of the cluster that was most recently allocated.
bFSInf_resvd	Reserved member.

Table 3: FAT Boot FS Info

3. Project Architecture

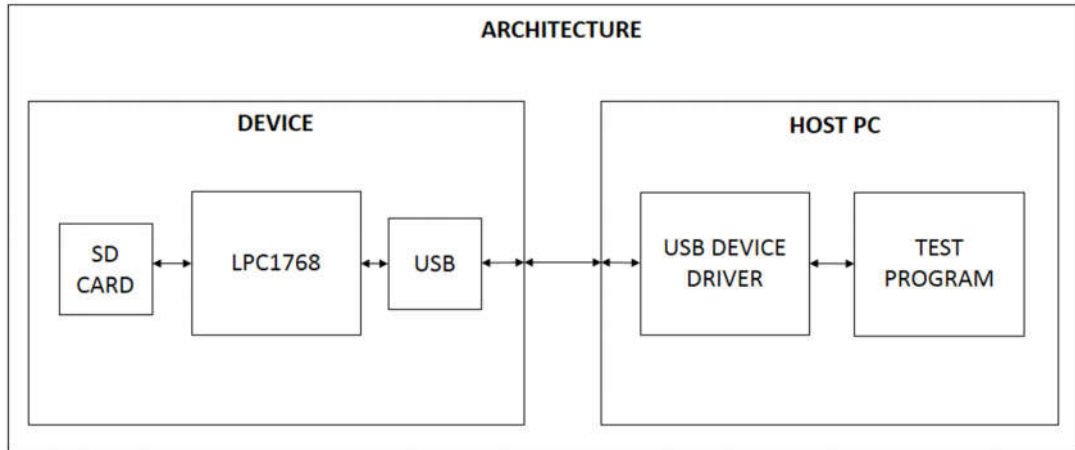


Figure 6: Project Architecture

Above figure represents the architecture of this project. Architecture consists of these parts

1. Device
 - a. SD Card
 - b. LPC1768
 - c. USB
2. Host PC
 - a. USB Device Driver
 - b. Test Program

Device:

Device consists of 3 sections those are SD Card, LPC1768 and USB sections. SD card is of 2GB memory size which is being formatted in FAT32 files system format. SD card is inserted in as SD Card adapter and that adapter is inserted in SD adapter slot of LPC1768 blueboard. This SD Card is interfaced to LPC1768 blueboard through SPI Protocol. LPC1768 blueboard is an ARM Cortex M3 board used in this project. USB section in device is the

interfacing of LPC1768 with USB Host controller on blueboard. The communication between device and host pc is done through USB only. This interfacing of USB and blueboard consists of following steps.

1. USB Hardware initialization.
2. USB Clock initialization.
3. USB Interrupt Initialization.

USB hardware initialization consists of USB register pin selection which are multiplexed in blueboard and pin direction selection of required registers used for USB.

USB clock initialization includes the enabling of Device clock which is used for USB operation and AHB clock which is used by USB bus for operations needed to bus.

USB Interrupt Initialization includes disabling of all interrupts first and then enabling the endpoint interrupts of USB, enabling USB main interrupt.

Host PC:

Host PC consists of the following sections

1. Linux USB Device driver
2. Test Program

Linux USB Device driver is developed through which the communication between device and PC will be done. This driver includes the detection of device, registration of device to USB Core, handling requests send by device and sending respective requests sent by user level test program.

Test program consists of read and write operations of the files to SD Cards. From this program user selects the read/write operation then the name of file is given after which selected operation on that file will be done through the device driver which is loaded to the Host PC OS which is already handling the device. Any operation by device can only be performed when it is initiated

from the user program. Device waits for the user program request and handles it. Test program is written in C.

4. Hardware Description

BLUEBOARD:

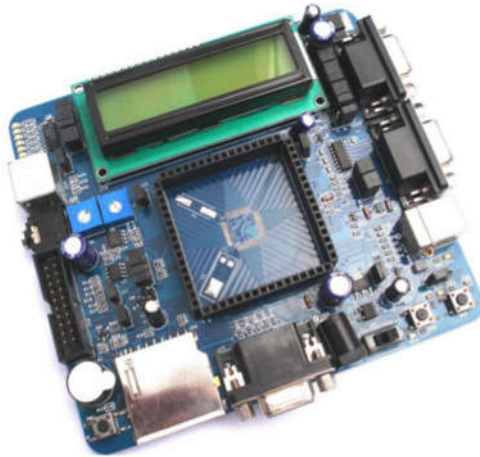


Figure 7: LPC1768 - Blueboard

• Introduction

Blueboard Base is modular approach to the ever evolving design of our flagship product Blueboard-LPC1768. It is designed keeping in mind the requirements of a development engineer, who could be working on different controllers but would prefer the same peripheral interface or a company working on different products using different controller and would not prefer to invest huge amount in individual boards or a hobby enthusiast who would be trying out different controllers but would not be willing to invest in different boards for each controller. For all these, the user can buy the required stamp module and work on this board.

Hardware:

- Dimensions of 114 X 127 mm²
- Two layer PCB (FR-4 material)
- Power supply: DC 6.5V with power LED
- On-board linear regulators generate +3.3V/500mA and +5v/500mA from power supply

- USB connector (as alternate power source)
- Extension headers for all microcontroller pins
- Two RS232 port connectors
- VGA connector
- PS/2 connector
- JTAG connector
- SD/MMC connector
- USB B-type connector with Link-LED
- All peripheral configurable via jumpers
- 256Kb I2C based EEPROM
- Audio power amplifier with audio jack
- 2 line X 16 character LCD with backlight control
- 8 controllable LEDs on SPI using 74HC595

USB HARDWARE

Features:

- Fully compliant with the USB 2.0 specification (full speed)
- Supports Soft Connect and Good Link features.
- Supports 32 physical (16 logical) endpoints.
- Supports Control, Bulk, Interrupt and Isochronous endpoints.
- Scalable realization of endpoints at run time.
- Endpoint maximum packet size selection (up to USB maximum specification) by software at run time.
- Supports DMA transfers on all non-control endpoints.

- Allows dynamic switching between CPU controlled and DMA modes.
- Double buffer implementation for Bulk and Isochronous endpoints.

Pin Description:

Name	Direction	Description
V _{BUS}	I	V _{BUS} status input. When this function is not enabled via its corresponding PINSEL register, it is driven HIGH internally.
USB_CONNECT	O	SoftConnect control signal.
USB_UP_LED	O	GoodLink LED control signal.
USB_D+	I/O	Positive differential data.
USB_D-	I/O	Negative differential data.

Table 3: USB Pin Configuration

Power Requirements

The USB protocol insists on power management by the device. This becomes very critical if the device draws power from the bus (bus-powered device). The following constraints should be met by a bus-powered device:

- A device in the non-configured state should draw a maximum of 100 mA from the bus.
- A configured device can draw only up to what is specified in the Max Power field of the configuration descriptor. The maximum value is 500 mA.
- A suspended device can draw a maximum of 2.5 mA.

Clocks:

Clock source	Description
AHB master clock	Clock for the AHB master bus interface and DMA
AHB slave clock	Clock for the AHB slave interface
usbclk	48 MHz clock from the dedicated USB PLL (PLL1) or the Main PLL (PLL0), used to recover the 12 MHz clock from the USB bus

Table 4: USB Clock

USB Registers:

USB Clock Control Registers(USBClkCtrl): This register controls the clocking of the USB Device Controller.

USB Device Interrupt Enable Register(USBDevIntEn): Writing a one to a bit in this register enables the corresponding bit in USBDevIntSt to generate an interrupt on one of the interrupt lines when set.

USB Device Interrupt Clear Register(USBDevIntClr): Writing one to a bit in this register clears the corresponding bit in USBDevIntSt. Writing a zero has no effect.

USB Device Interrupt Set Register(USBDevIntSet): Writing one to a bit in this register sets the corresponding bit in the USBDevIntSt. Writing a zero has no effect. USBDevIntSet is a write-only register.

USB Endpoint Interrupt Enable Registers(USBEPIntEp): Setting a bit to 1 in this register causes the corresponding bit in USBEPIntSt to be set when an interrupt occurs for the associated endpoint. Setting a bit to 0 causes the corresponding bit in USBDMARSt to be set when an interrupt occurs for the associated endpoint. USBEPIntEn is a read/write register.

USB Endpoint Interrupt Set Register(USBEPIntSt): Writing a one to a bit in this register sets the corresponding bit in USBEPIntSt. Writing zero has no effect. Each endpoint has its own bit in this register. USBEPIntSet is a write-only register.

USB Receive Data Registers: For an OUT transaction, the CPU reads the endpoint buffer data from this register. Before reading this register, the RD_EN bit and LOG_ENDPOINT field of the USBCtrl register should be set appropriately. On reading this register, data from the selected endpoint buffer is fetched. The data is in little endian format: the first byte received from the

USB bus will be available in the least significant byte of USBRxData. USBRxData is a read-only register.

USB Transmit Data Register: For an IN transaction, the CPU writes the endpoint data into this register. Before writing to this register, the WR_EN bit and LOG_ENDPOINT field of the USBCtrl register should be set appropriately, and the packet length should be written to the USBTxPlen register. On writing this register, the data is written to the selected endpoint buffer. The data is in little endian format the first byte sent on the USB bus will be the least significant byte of USBTxData. USBTxData is a write-only register.

USB Control Register(USBCtrl): This register controls the data transfer operation of the USB device.

USB Command Code Registers: This register is used for sending the command and write data to the SIE.

USB Command Data Register: This register contains the data retrieved after executing a SIE command.

Synchronous Serial Port:

- **Basic Configuration**
- **Power:** In the PCONP register, set bit PCSSP0 to enable SSP0 and bit PCSSP1 to enable SSP1.
- On reset, both SSP interfaces are enabled (PCSSP0/1 = 1).
- **Clock:** In PCLKSEL0 select PCLK_SSP1; in PCLKSEL1 select PCLK_SSP0.
- In master mode, the clock must be scaled down.
- **Pins:** Select the SSP pins through the PINSEL registers and pin modes through the PINMODE registers.

- **Interrupts:** Interrupts are enabled in the SSP0IMSC register for SSP0 and SSP1IMSC register for SSP1. Interrupts are enabled in the NVIC using the appropriate Interrupt Set Enable register.
- **Initialization:** There are two control registers for each of the SSP ports to be configured: SSP0CR0 and SSP0CR1 for SSP0, SSP1CR0 and SSP1CR1 for SSP1.

Features:

- Compatible with Motorola SPI, 4-wire TI SSI, and National Semiconductor Microwirebuses.
- Synchronous Serial Communication.
- Master or slave operation.
- 8 frame FIFOs for both transmit and receive.
- 4 to 16 bit data frame.
- DMA transfers supported by GPDMA.

SD-Card

SD Pin	miniSD Pin	microSD Pin	Name	I/O	Logic	Description
1	1	2	DAT3	I/O	PP	SD Serial Data 3
2	2	3	CMD	I/O	PP, OD	Command, Response
3	3		VSS	S	S	Ground
4	4	4	VDD	S	S	Power
5	5	5	CLK	I	PP	Serial Clock
6	6	6	VSS	S	S	Ground
7	7	7	DAT0	I/O	PP	SD Serial Data 0
8	8	8	DAT1 nIRQ	I/O O	PP OD	SD Serial Data 1 (memory cards) Interrupt Period (SDIO cards share pin via protocol)
9	9	1	DAT2	I/O	PP	SD Serial Data 2

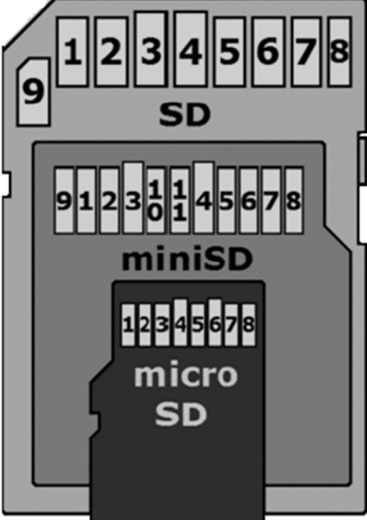


Figure 8: SD Card Pin Description

5. Software Requirements and Specifications

In this project C programming language is used in all sections. Following libraries are used for interfacing of SD card and USB in LPC1768.

Tool Chain:

- ARM tool chain used for firmware is '**gcc-arm-none-eabi-5_4-2016q2**' in which '**arm-none-eabi-as**' is assembler and '**arm-none-eabi-gcc**' is the compiler
- For test program compilation '**gcc c compiler**' is used

SD Card Library:

Since SD Card is formatted in FAT32 file system, a FAT32 IO library was used for operations on SD Card. This library can write a file into SD Card character by character. This library can perform following operation on SD Card.

1. Open File
2. Read File
3. Write File
4. Close File
5. List out the files present in SD Card
6. Memory statistics of SD Card

USB Stack:

For USB implementation in LPC1768 Bertrik Sikken stack was used in device. This stack is a LPC1768 series board specific USB stack. This stack consists of following things

1. USB Hardware Initialization of LPC1768
2. USB Endpoint ISR registrations

3. USB Device descriptors
4. USB Clock enabling
5. Reading of Data from Host
6. Writing of Data to host

This stack can send a 64Bytes of packet at a time or receive at a time using endpoints.

Device Driver:

Device driver module for host PC is written in C language. 'usb.h' is major header file used for implementation of USB Device driver in PC. This driver can handle the requests for the device with given vendor ID and Product ID. This driver can perform following operations on device.

1. Probe
2. Disconnect
3. Open
4. Close
5. Read
6. Write

Test Program:

Test program is the main application from which device operations are done. This is written in C language. This program will do operations on '/dev/pendrive_driver' file which. Following operations can be performed by this test program.

1. Read File
2. Write File
3. List Files
4. Copy File to host PC.

6. Source Code Explanation

Device Source code explanation:

USB_device.c:

USB_device.c file is the main firmware file of device. Following array is the USB Descriptor which is used by device for configuration of USB.

```
static const unsigned char abDescriptors[] = {
    /* Device descriptor */
    0x12,                // Length of descriptor
    DESC_DEVICE,         // Type of descriptor
    LE_WORD(0x0200),     // bcdUSB
    0x00,                // bDeviceClass
    0x00,                // bDeviceSubClass
    0x00,                // bDeviceProtocol
    MAX_PACKET_SIZE0,   // bMaxPacketSize
    LE_WORD(0xABCD),     // idVendor
    LE_WORD(0x2017),     // idProduct
    LE_WORD(0x0100),     // bcdDevice
    0x01,                // iManufacturer
    0x02,                // iProduct
    0x03,                // iSerialNumber
    0x01,                // bNumConfigurations
    /* Configuration Descriptor */
    0x09,
    DESC_CONFIGURATION,
    LE_WORD(0x27),       // wTotalLength
    0x01,                // bNumInterfaces
    0x01,                // bConfigurationValue
    0x00,                // iConfiguration
    0x80,                // bmAttributes //bus powered
    0x32,                // bMaxPower (100mA Current)
    /* Interface Descriptor */
    0x09,
    DESC_INTERFACE,
    0x00,                // bInterfaceNumber
    0x00,                // bAlternateSetting
    0x03,                // bNumEndPoints
    0xFF,                // bInterfaceClass
    0x00,                // bInterfaceSubClass
    0x00,                // bInterfaceProtocol
    0x00,                // iInterface
    /* Bulk IN 1 Endpoint */
    0x07,
    DESC_ENDPOINT,
    BULK_IN_1_EP,        // bEndpointAddress
    0x02,                // bmAttributes = Bulk
    LE_WORD(MAX_PACKET_SIZE), // wMaxPacketSize
    0x00,                // bInterval
    /* Bulk IN 2 Endpoint */
    0x07,
```

```

DESC_ENDPOINT,
BULK_IN_2_EP,          // bEndpointAddress
0x02,                  // bmAttributes = Bulk
LE_WORD(MAX_PACKET_SIZE), // wMaxPacketSize
0x00,                  // bInterval
/* Bulk OUT 1 Endpoint*/
0x07,
DESC_ENDPOINT,
BULK_OUT_1_EP,         // bEndpointAddress
0x02,                  // bmAttributes = Bulk
LE_WORD(MAX_PACKET_SIZE), // wMaxPacketSize
0x00,                  // bInterval
/* string descriptors */
0x04,
DESC_STRING,
LE_WORD(0x0409),
// manufacturer string
0x18,
DESC_STRING,
'D', 0, 'E', 0, 'S', 0, 'D', 0, ' ', 0, 'A', 0, 'U', 0, 'G', 0, 'I', 0, '7', 0, ' ', 0,
// product string
0x10,
DESC_STRING,
'U', 0, 'S', 0, 'B', 0, ' ', 0, 'A', 0, 'R', 0, 'M', 0,
// serial number string
0x12,
DESC_STRING,
'0', 0, '0', 0, '0', 0, '0', 0, '0', 0, '0', 0, '7', 0,
// terminator
0
};

```

This descriptor is an unsigned character array which has sub sections as device descriptor, Configurations Descriptor, Interface Descriptor, Endpoints, string descriptors in which members are represented and follows.

- Every descriptor's first member's is length of descriptor
- Device descriptor consists of 12 members in which it represent the type of device, class of device, Maximum Packet size, Product ID, Vendor ID and many.
- Configuration Descriptor is of 9 members which gives info likes word length, number of interfaces here it is 1, power type – here it is bus powered, maximum current – here it is 100mA
- Interface Descriptor is of 9 members which provides info of number of endpoints, interface class, interface protocol and interface number.
- Endpoint descriptor is of length 7 Bytes which has endpoint address, maximum packet size of that endpoint, type of endpoint

- String descriptor has 1st members as length of string and the string in which each character is separated by '0'. This Device descriptor is terminated by '0'

main():

```
returnStatus = SD_Init(&sdcardType);
```

This SD_init() function initiates the SD Card which is PINS of SD Card SPI interfacing, checks SD Card file system, checks whether card is properly inserted or not.

```
USBInit();
USBRegisterDescriptors(abDescriptors);
USBHwRegisterEPIntHandler(BULK_IN_1_EP, BulkIn);
USBHwRegisterEPIntHandler(BULK_OUT_1_EP, BulkOut);
USBHwNakIntEnable(INACK_BI);
DBG("Starting USB communication\n");
NVIC_EnableIRQ( USB_IRQn );
DBG("Connecting to USB bus\n");
USBHwConnect(TRUE);
```

USB_init() initializes USB Hardware and USB clocks and interrupts of USB. After that device descriptor is registered, Endpoint ISRs are registered, bulk_in interrupts are enabled, USB interrupt is enabled, USB is connected to bus.

```
static void BulkIn(unsigned char bEP, unsigned char bEPStatus)
static void BulkOut(unsigned char bEP, unsigned char bEPStatus)
```

These two are ISRs of BULK_IN and BULK_OUT interrupts which are called by Interrupt handler when that interrupt is occurred due to the respective request. In this ISRs the read and write operations to SD Card are done.

SD Card:

```
fileConfig_st* FILE_Open(char* filename,uint8_t fileOperation,uint8_t *fileOpenSts);
void FILE_Close(fileConfig_st *ptr);
uint8_t FILE_Delete(char *fileName);
char FILE_GetCh(fileConfig_st *filePtr);
void FILE_PutCh(fileConfig_st *filePtr, char data);
uint8_t FILE_GetList (fileInfo *fileList);
void FILE_GetMemoryStatics(uint32_t *ptr_totalMemory, uint32_t *ptr_freeMemory);
```

These functions are used in SD Card FAT32 library. These functions description is as follows.

- FILE_Open() has three parameters as file name, mode (read/write/append), return status
- FILE_Close() has file pointer as parameter which closes the file pointing by that pointer in SD Card.
- FILE_Delte() deletes the file from SD Card by the name of file given in parameter
- FILE_GetCh() is used to read a character from a file pointed by pointer given in parameter from SD Card
- FILE_PutCh() is used to write a character to a file pointed by the pointer given in parameter to SD Card
- FILE_GetList() returns the file structure by traversing all files
- FILE_GetMemoryStatics() returns the total size of SD Card and used memory in SD Card

Driver:

```

static int __init pendrive_init(void)
static int pendrive_probe(struct usb_interface *interface, const struct usb_device_id *id);
static int pendrive_open(struct inode *inode, struct file *file);
static ssize_t pendrive_read(struct file *file, char *buffer, size_t count, loff_t *fpos);
static ssize_t pendrive_write(struct file *file, const char *user_buffer, size_t count, loff_t *fpos);
static void pendrive_delete(struct kref *kref);
static int pendrive_close(struct inode *inode, struct file *file);
static void pendrive_disconnect(struct usb_interface *interface);
static void __exit pendrive_exit(void)

```

Above functions are implemented in USB Driver whose description is as follows

- pendrive_init() is called when module is inserted.
- pendrive_probe() is called by USB core when device is inserted in which local structure, interface structure's memory is allocated using kzalloc() and endpoint addresses where found by traversing endpoint array presented in interface structure.
- pendrive_open() is called when open() system call is called on driver by user program. In this the buffer to which data is written, it's pointer is stored in private_data pointer presented in file structure
- pendrive_read() is called when read() system call is called on driver by user program. In this function the data is sent to user program from IN Endpoint by its address by using following function

```

/* Read data from Bulk end point */
//usb_bulk_msg will create an URB and send it to the specified device in its arg and wait

```

```
//to complete before it returning to the caller
retval=usb_bulk_msg(dev->udev, //pointer to usb device to send bulk message
usb_rcvbulkpipe(dev->udev, dev->bulk_in_endpointAddr), //receives data from bulk_in_endpoint
dev->bulk_in_buffer, //buffer where data is stored after reading from endpoint
MIN(dev->bulk_in_size, count), //no of bytes to be read
&read_count, //address of count variable
HZ*10); //timeout
```

- pendrive_write() is called when write() system call is called on driver by user program. In this function the data sent from user program is written to OUT Endpoint by its address using following function

```
/* Write data to bulk end point */
retval=usb_bulk_msg(dev->udev, //pointer to usb device to send bulk message
usb_sndbulkpipe(dev->udev, dev->bulk_out_endpointAddr), //receives data from bulk_out_endpoint
write_buf, //write buffer
write_count, //no of bytes to write
&write_count, //address of count variable
HZ*10); //timeout
```

- pendrive_close() is called when close() system call is done on driver by user program. In this the local buffer which is allocated in pendrive_probe() is deallocated
- pendrive_disconnect() is called by USB Core when device is detached from Host PC in which interface structure and other structures were deallocated.
- pendrive_exit() is called by OS when module is removed.

Test Program:

```
fd=open("/dev/pendrive_driver", O_RDWR);
```

open() system called is called by passing path of driver file by which the driver file is opened in Read and Write Mode and file descriptor of opened file is returned.

```
printf("\nSelect operation:\n");
printf("0. EXIT\n");
printf("1. Write FILE\n");
printf("2. Read FILE\n");
printf("3. List Files\n");
printf("4. Copy a file from SD Card\n");
printf(">>");
scanf("%d", &opt);

switch(opt)
{
    case 0 : /** EXIT **/
        break;
```

```

        case 1 : /** Write File operation **/
            break;

        case 2 : /** READ File operation **/
            break;

        case 3 : /** LIST Files operation **/
            break;

        case 4 : /** Copy File from SD Card **/
            break;

        default:break;
    }

```

- In test program there are 4 operations namely write, read, list files, copy file from SD Card.
- In write operation steps:
 1. Read new File name of maximum 11 character from user
 2. Write 'WRITE' string to device
 3. Write name of file to device
 4. Read character by character from user untill EOF('>')
 5. Write the character into 'new_filebuf',if new_filebuf fills then write the new_filebuf to device
 6. If EOF reached by USER in less than 64 bytes, then allocate user entered number of bytes+1 to temp_out_string, write all that bytes to temp_out_string then write that temp_out_string to device, free buffer.
 7. Sleep for 1 second
 8. Write '___END___' string to device
 9. Sleep for 1 second
- In Read operation steps:
 1. Read file name to be read from user
 2. Write 'READ' string to device
 3. Sleep for 1 second
 4. Write file name to device
 5. Read 64 bytes repeatedly untill recived data is '___END___' and print it
- In List file operation steps:
 1. Write "LIST" string to device
 2. Read from device and print it which is file name

3. If read string is ‘__LIST__END__’ then list is ended

- In copy file from SD card operation:
 1. Read file name to be read from user
 2. Write ‘READ’ string to device
 3. Sleep for 1 second
 4. Write file name to device
 5. Create a file in host with same name of reading file in open it in write mode
 6. Read 64bytes data repeatedly until received data is ‘__END__’ and write the received data to opened file in Host
 7. Close file in Host

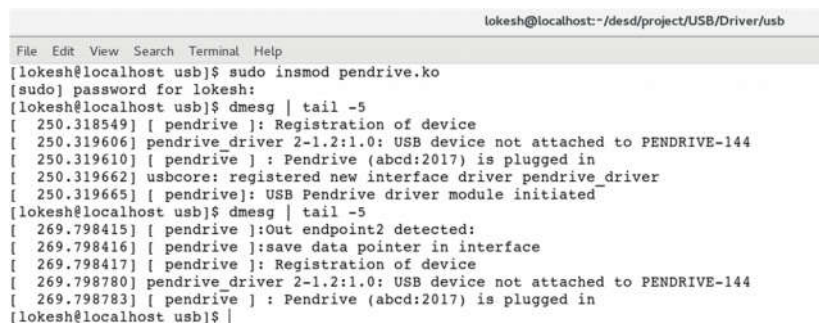
7. Testing

Test Procedures:

- Driver module is inserted in host PC and tested in 'dmesg'.
- Device is inserted, then the driver file creation is checked in '/dev' using 'ls' command.
- Test program executable file was run and options were selected by user based on his operations needed
- Device is unplugged then the status was checked in 'dmesg'
- Screenshots of all testing procedure were provided below

Screenshots

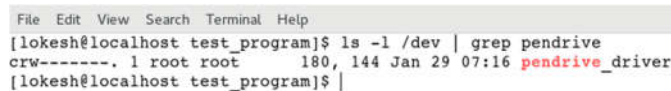
Driver Module insertion:



```
lokesh@localhost: ~/desd/project/USB/Driver/usb
File Edit View Search Terminal Help
[lokesh@localhost usb]$ sudo insmod pendrive.ko
[sudo] password for lokesh:
[lokesh@localhost usb]$ dmesg | tail -5
[ 250.318549] [ pendrive ]: Registration of device
[ 250.319606] pendrive driver 2-1.2:1.0: USB device not attached to PENDRIVE-144
[ 250.319610] [ pendrive ] : Pendrive (abcd:2017) is plugged in
[ 250.319662] usbcore: registered new interface driver pendrive_driver
[ 250.319665] [ pendrive]: USB Pendrive driver module initiated
[lokesh@localhost usb]$ dmesg | tail -5
[ 269.798415] [ pendrive ]:Out endpoint2 detected:
[ 269.798416] [ pendrive ]:save data pointer in interface
[ 269.798417] [ pendrive ]: Registration of device
[ 269.798780] pendrive driver 2-1.2:1.0: USB device not attached to PENDRIVE-144
[ 269.798783] [ pendrive ] : Pendrive (abcd:2017) is plugged in
[lokesh@localhost usb]$
```

Figure 9: Driver Module Insertion

Creation of driver file in /dev:



```
File Edit View Search Terminal Help
[lokesh@localhost test_program]$ ls -l /dev | grep pendrive
crw----- 1 root root      180, 144 Jan 29 07:16 pendrive_driver
[lokesh@localhost test_program]$
```

Figure 10: Driver file creation

User Test Program execution:

```
File Edit View Search Terminal Help
[lokesh@localhost test_program]$ sudo ./a.out

Select operation:
0. EXIT
1. Write FILE
2. Read FILE
3. List Files
4. Copy a file from SD Card
>>|
```

Figure 11: Test program execution

Writing a file to SD Card:

```
File Edit View Search Terminal Help
[lokesh@localhost test_program]$ sudo ./a.out

Select operation:
0. EXIT
1. Write FILE
2. Read FILE
3. List Files
4. Copy a file from SD Card
>>1
New FILE name(11characters MAX): TEST.TXT
'TEST.TXT' is given File Name
'WRITE' command sent to Device
'TEST.TXT' filename sent to Device
Enter data ['>' to EOF]: tHis is a test file written to Sd card>
Total data written to SD Card
__END__ sent to device

Select operation:
0. EXIT
1. Write FILE
2. Read FILE
3. List Files
4. Copy a file from SD Card
>>|
```

Figure 12: Write a file to SD Card

Reading File from SD Card:

```
Select operation:
0. EXIT
1. Write FILE
2. Read FILE
3. List Files
4. Copy a file from SD Card
>>2
File Name to read: TEST.TXT
'TEST.TXT' is given File Name
'READ' command sent to Device
press any key to continue...

'TEST.TXT' file name sent to Device
'READ OPERATION' started...
tHis is a test file written to Sd card
__EOF__

Select operation:
0. EXIT
1. Write FILE
2. Read FILE
3. List Files
4. Copy a file from SD Card
>>|
```

Figure 13: Reading file from SD Card

List files operation:

```
Select operation:
0. EXIT
1. Write FILE
2. Read FILE
3. List Files
4. Copy a file from SD Card
>>3
'LIST' command sent to Device
LIST Read operation Initiated..

LISTFAT32.
TEMP.TXT
LOK.TXT
BIT.TXT
LOKI.TXT
WHY.TXT
ONE.TXT
T[1]
__LIST_ENDED__

Select operation:
0. EXIT
1. Write FILE
2. Read FILE
3. List Files
4. Copy a file from SD Card
>>|
```

Figure 14: List file operation

Copy file from SD Card:

```
[lokesh@localhost test_program]$ sudo ./a.out

Select operation:
0. EXIT
1. Write FILE
2. Read FILE
3. List Files
4. Copy a file from SD Card
>>4
File Name to COPY: TEST.TXT
'TEST.TXT' is given File Name
'READ' command sent to Device
press any key to continue...

'TEST.TXT' file name sent to Device
copying..
__EOF__

Select operation:
0. EXIT
1. Write FILE
2. Read FILE
3. List Files
4. Copy a file from SD Card
>>0
[lokesh@localhost test_program]$ cat TEST.TXT
tHis is a test file wrtten to Sd card[lokesh@localhost test_program]$
```

Figure 15: Copy file from SD Card

Device unplugged and complete 'dmesg' from insertion to unplug of device:

```
lokesh@localhost:~  
File Edit View Search Terminal Help  
[lokesh@localhost ~]$ dmesg | tail -30  
[ 448.746932] usb 2-1.2: Product: USB ARM  
[ 448.746937] usb 2-1.2: Manufacturer: DESD AUG17.  
[ 448.746941] usb 2-1.2: SerialNumber: 0000007  
[ 448.748559] [ pendrive ]:In endpoint0 detected:  
[ 448.748562] [ pendrive ]:Out endpoint2 detected:  
[ 448.748563] [ pendrive ]:save data pointer in interface  
[ 448.748565] [ pendrive ]: Registration of device  
[ 448.748754] pendrive_driver 2-1.2:1.0: USB device not attached to PENDRIVE-144  
[ 448.748757] [ pendrive ] : Pendrive (abcd:2017) is plugged in  
[ 466.398510] [ pendrive ]: pendrive_close() called  
[ 466.398522] [ pendrive ]: Bulk in Buffer freed  
  
[ 466.398524] [ pendrive ]: Device local structure freed  
  
[ 468.927328] [ pendrive ]: pendrive_open() called  
[ 468.927344] [ pendrive ]: retval: 0  
  
[ 481.348752] [ pendrive ]: Pendrive read data  
[ 481.348760] [ pendrive ]: copy_to_user() done  
[ 481.348936] [ pendrive ]: Pendrive read data  
[ 481.348940] [ pendrive ]: copy_to_user() done  
[ 485.852463] [ pendrive ]: pendrive_close() called  
[ 527.504177] perf: interrupt took too long (3919 > 3916), lowering kernel.perf_event_max_sample_rate to 51000  
[ 607.208175] usb 2-1.2: USB disconnect, device number 9  
[ 607.210816] [ pendrive ]: Bulk in buffer freed  
  
[ 607.210822] [ pendrive ]: Device local structure freed  
  
[ 607.210830] [ pendrive ]: Pendrive is unplugged - MINOR : 144  
[ 607.210834] Pendrive unplugged  
[lokesh@localhost ~]$ |
```

Figure 16: Total dmesgs of process

8. Future Scope

- Writing and Reading of audio files can be added.
- Mobile support can be added so it can be detected in mobile similar to PC using OTG

9. Conclusion

Using LPC1768 Blueboard a file is read and written to SD Card from PC successfully through USB from user test program.

10. References

1. FAT32 File system by Maverick
2. FAT32 File IO library reference from Exploreembedded
3. USB
4. Linux Device Drivers by Alessandro Rubini
5. Linux Kernel Source Code
6. An Introduction to USB Bus 2.0 – Cypress Semiconductor