ISSN 2047-3338

A Frame Work for Virtual USB Devices under Linux Environment

M. A. Naeem[1] and Rizwan Khan[2]
[1,2]Department of Computer Science, The University of Auckland, New Zealand

***Abstract*— In industry firmware development for USB devices is very demanding. The objective of this research work is to develop a framework for writing and testing the firmware for USB devices even before the devices are manufactured. Faulty hardware is often a bottleneck in firmware development and time is often lost due to this dilemma. There is need for a framework that could reliably test the firmware without the need of actual hardware. Minimizing the time to sell a product is the key to success in today's growing market. With the development of this framework, hardware and software development can be done in parallel. Such a frame work can be developed on any operating system like windows, Linux etc. In this research work, we have selected Linux operating system for the implementation of Virtual Device Frame work due to its open source nature.**

***Index Terms*— Framework, USB, Linux, Hardware and Testing**

I. INTRODUCTION

THE Universal Serial Bus (USB) is a fast and flexible interface for connecting devices to computers. Every new PC has at least a couple of USB ports. The interface is versatile enough to use with standard peripherals like keyboards and disk drives as well as more specialized devices, including one-of-a-kind designs. In short, USB is very different from the legacy interfaces it's replacing. A USB device [8] may use any of four transfer types and three speeds. On attaching to a PC, a device must respond to a series of requests that enable the PC to learn about the device and establish communications with it. In the PC, every device must have a low-level driver to manage communications between applications and the system's USB drivers [9]. Developing a USB device and the software that communicates with it requires knowing something about how USB works and how the PC's operating system implements the interface.

USB communications takes place between the host and endpoints located in the peripherals. An endpoint is a uniquely addressable portion of the peripheral that is the source or receiver of data. Four bits define the device's endpoint address; codes also indicate transfer direction and whether the transaction is a "control" transfer. Endpoint 0 is reserved for control transfers, leaving up to 15 bi-directional destinations or sources of data within each device [1].

The idea of endpoints leads to an important concept in USB transactions, that of the pipe. All transfers occur through virtual pipes that connect the peripheral's endpoints with the host. When establishing communications with the peripheral, each endpoint returns a descriptor, a data structure that tells the host about the endpoint's configuration and expectations. Descriptors include transfer type, max size of data packets, perhaps the interval for data transfers, and in some cases, the bandwidth needed. Given this data, the host establishes connections to the endpoints through virtual pipes, which even have a size (bandwidth), to make them analogous to household plumbing [3].

USB supports four data transfer types: control, isochronous, bulk, and interrupt.

Control transfers exchange configuration, setup, and command information between the device and the host. CRCs check the data and initiate retransmissions when needed to guarantee the correctness of these packets.

Bulk transfers move large amounts of data when timely delivery isn't critical. Typical applications include printers and scanners. Bulk transfers are fillers, claiming unused USB bandwidth when nothing more important is going on. CRCs protect these packets.

Finally, isochronous transfers handle streaming data like that from an audio or video device. It is time sensitive information so, within limitations, it has guaranteed access to the USB bus. No error checking occurs so the system must tolerate occasional scrambled bytes [6], [7].

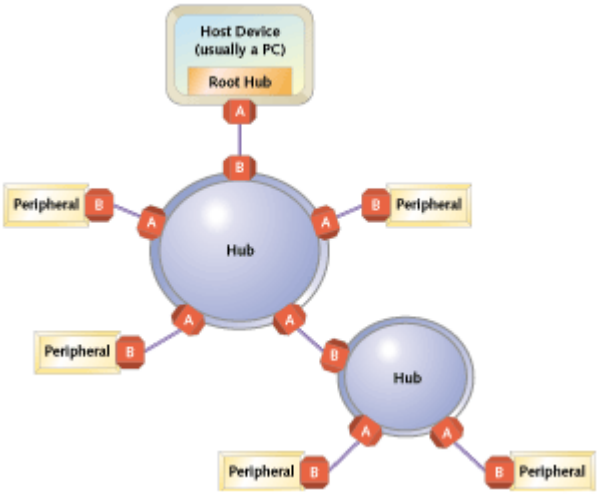

Fig. 1. USB Hub Architecture

In industry firmware development for USB devices is very demanding. The objective of this project is to develop a framework for writing and testing the firmware for USB devices even before the devices are manufactured.

Faulty hardware is often a bottleneck in firmware development and time is often lost due to this dilemma. There is need for a framework that could reliably test the firmware without the need of actual hardware. Time is money. Minimizing the time to sell a product is the key to success in today's growing market. With the development of this a framework, hardware and software development can be done in parallel.

Such a frame work can be developed on any operating system like windows, Linux [5] etc. In this research we have selected Linux operating system [4] for the implementation of Virtual Device Frame work. The main reason for this selection is the open source nature of Linux. Also there are many classes for USB devices. One of them is Mass Storage class [12]. Currently, the scope of project is limited to USB Mass Storage Class only.

The rest of the paper is organized as: In section II, we discussed the background history and related work of the research. In section III, we presented our proposed system design and architecture. We conclude our work in section IV.

II. BACKGROUND

A. The USB Mass Storage Class

This section gives an overview of the USB Mass Storage Class [12] specification overview. How mass storage devices behave on the USB bus is the subject of this and other USB Mass Storage Class specifications. In addition to this Overview specification, several other USB Mass Storage Class specifications are supported by the USB Mass Storage Class Working Group (CWG). The titles of these specifications are:

- USB Mass Storage Class Control/Bulk/Interrupt (CBI) Transport
- USB Mass Storage Class Bulk-Only Transport
- USB Mass Storage Class UFI Command Specification
- USB Mass Storage Class Bootability Specification
- USB Mass Storage Class Compliance Test Specification
- The USB Mass Storage Class Control/Bulk/Interrupt

CBI Transport specification is approved for use only with full-speed floppy disk drives. CBI shall not be used in high-speed capable devices, or in devices other than floppy disk drives. Usage of CBI for any new design is discouraged.
Note: The bootability and Compliance Test specifications are still under development, and are not yet publicly available.

1). Specification Relationships

The CBI and Bulk-Only specifications are each intended to be stand-alone documents for the USB Mass Storage class, enabling development of a USB Mass Storage compliant device. A device manufacturer may choose to implement both CBI and Bulk-Only, but shall follow each specification as applicable. Booting an operating system from a USB Mass Storage Class device requires no special considerations with regard to Mass Storage Class support. Either CBI or Bulk-Only devices may be bootable. Bootability may, however, require other considerations such as particular types of media formatting, etc. Such considerations are hardware- or operating system dependent, and are beyond the scope of the Mass Storage Class specifications.

SubClass Code	Command Block Specification	Comment
01h	Reduced Block Commands (RBC) T10 Project 1240-D	Typically, a Flash device uses RBC command blocks. However, any Mass Storage device can use RBC command blocks.
02h	SFF-8020i, MMC-2 (ATAPI)	Typically, a C/DVD device uses SFF-8020i or MMC-2 command blocks for its Mass Storage interface.
03h	QIC-157	Typically, a tape device uses QIC-157 command blocks.
04h	UFI	Typically a floppy disk drive (FDD) device
05h	SFF-8070i	Typically, a floppy disk drive (FDD) device uses SFF-8070i command blocks. However, an FDD device can be in another subclass (for example, RBC) and other types of storage devices can belong to the SFF-8070i subclass.
06h	SCSI transparent command set	
07h – FFh	Reserved for future use.	

Fig. 2. SubClass Codes Mapped to Command Block Specifications

2). Mass Storage Subclass

The Interface Descriptor of a USB Mass Storage Class (Fig. 2) device includes a bInterfaceSubClass field. This field denotes the industry-standard protocol transported by a Mass Storage Class interface. The value of the bInterfaceSubClass field shall be set to one of the Subclass codes as shown in the following table. Note that the Subclass code values used in the bInterfaceSubClass field specify the industry-standard specification that defines transport protocols and command code systems transported by the interface; these Subclass codes do not specify a type of storage device (such as a CD-ROM or floppy disk drive).

B. Linux USB Architecture

In Linux there exists a subsystem called ``The USB Core'' with a specific API to support USB devices and host controllers. Its purpose is to abstract all hardware or device dependent parts by defining a set of data structures, macros and functions. The USB core contains routines common to all USB device drivers and host controller drivers. These functions can be grouped into an upper and a lower API layer. There exists an API for USB device drivers and another one for host controllers. The following section concentrates on the USB device driver layer, because the development for host controller drivers is already finished. This section will give an overview of the USB framework by explaining entry points and the usage of API functions. The Fig. 3 shows the architecture of Linux USB [2], [5], [14].

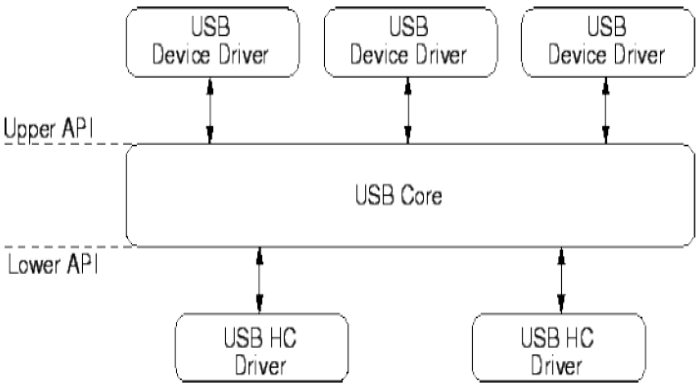

Fig. 3. Linux USB Architecture

1). Framework Data Structures

USB devices drivers are registered and deregistered at the subsystem. A driver must register 2 entry points and its name. For specific USB devices (which are not suitable to be registered at any other subsystem) a driver may register a couple of file operations and a minor number. In this case the specified minor number and the 15 following numbers are assigned to the driver. This makes it possible to serve up to 16 similar USB devices by one driver. The major number of all USB devices is 180.

```
struct usb_driver {
const char *name;

void * (*probe)(struct usb_device *, unsigned int,
const struct usb_device_id *id_table);
void (*disconnect)(struct usb_device *, void *);

struct list_head driver_list;

struct file_operations *fops;
int minor;
struct semaphore serialize;
int (*ioctl) (struct usb_device *dev, unsigned int code,
void *buf);
const struct usb_device_id *id_table;
};
```

Fig. 4. Framework Data structures

- *name:* Usually the name of the module.
- *probe:* The entry point of the probe function.
- *disconnect:* The entry point of the disconnect function.
- *driver_list:* For internal use of the subsystem - initialize to {NULL,NULL}
- *fops:* The usual list of file operations for a driver
- *minor:* The base minor number assigned to this device (the value has to be a multiple of 16)
- *serialize:*
- *ioctl:*
- *id_table:*

III. PROPOSED SYSTEM DESIGN

A. File-backed Storage

The File-backed Storage (FS) provides support for the USB Mass Storage class. It can appear to a host as a set of up to 8 SCSI disk drives (called Logical UNits or LUNs), although most of the time a single LUN is all you will need. The information stored for each LUN must be maintained by the gadget somewhere, either in a normal file or in a block device such as a disk partition or even a ramdisk.

This file or block device is called the backing storage for the gadget, and tell FSG where the backing storage is when gadget driver is loaded:

bash# modprobe g_file_storage file=/root/data/backing_file

This command tells FSG to provide a single LUN with backing storage maintained in /root/data/backing_file. If two LUNs are required, where the second LUN used /dev/hda7 as its backing storage, you would do:

bash# modprobe g_file_storage
file=/root/data/backing_file,/dev/hda7

Under Linux 2.6 [4], [15]; if "removable=y" is added to the modprobe line then FSG will act like a device with removable media and allow to specify the backing storage using sysfs attributes. In fact, by doing this the "file=..." parameter can be omited entirely. The gadget will resemble a ZIP drive with no cartridge inserted until sysfs is used to specify some backing storage.

AN IMPORTANT WARNING! While FSG is running and the gadget is connected to a USB host, that USB host will use the backing storage as a private disk drive. It will not expect to see any changes in the backing storage other than the ones it makes. Extraneous changes are liable to corrupt the filesystem and may even crash the host. Only one system (normally, the USB host) may write to the backing storage, and if one system is writing that data, no other should be reading it. The only safe way to share the backing storage between the host and the gadget's operating system at the same time is to make it read-only on both sides.

B. Creating A Backing Storage File

Backing storage requires some preparation before FSG can use it. To start with, if the backing storage is a regular file then the file must be created beforehand, with its full desired size. (FSG won't create a backing storage file and won't change the size of an existing file.) In the example above, if /root/data/backing_file is wanted to represent a 64MB drive then it should be created using a command something like this:

bash# dd bs=1M count=64 if=/dev/zero
of=/root/data/backing_file
64+0 records in
64+0 records out

This has to be done before one can load g_file_storage, but it only has to be done once. If the backing storage is a block device or disk partition such as /dev/hda7 then one don't have to create it beforehand, because it will already exist.

C. Partitioning the Backing Storage

However, creating the backing storage isn't enough. It's like having a raw disk drive; which needs partition and file system before using it. (Strictly speaking there is no need to partition it. The entire drive can be treated as a single large device, like a floppy disk. This will be confusing, though, and some versions of Windows won't work with an unpartitioned USB drive). To partition the backing storage a partition table has to be created by using the fdisk program. Here's an example showing how to do it.

The example assumes that gadget will be used with a Windows host. It's a little tricky because fdisk needs help when working with something other than an actual device. Begin by starting up fdisk and telling it the name of backing storage. A message something like this: bash# fdisk /root/data/backing_file will be received. Device contains neither a valid DOS partition table, nor Sun or SGI disklabel. Build a new DOS disklabel. Changes will remain in memory only, until you decide to write them. After that, of course, the previous content won't be recoverable.

D. Heads, Sectors and Cylinders

As fdisk needs to set the heads, sectors, and cylinders values. (Some versions only need to set the number of cylinders, but they're wrong. It is baecause of miscalculation of the size of backing file; as default values are used and ignoring the actual file size.) The numbers are somewhat arbitrary; the scheme shown here works good. Give the "x" (eXpert or eXtra) command: Command (m for help): x

Then number of sectors/track will be set. g_file_storage uses a sector size of 512 bytes, so 8 sectors/track will give 4096 bytes per track. This is good because it matches the size of a memory page (on a 32-bit processor). Expert command (m for help): s
Number of sectors (1-63): 8
Warning: setting sector offset for DOS compatiblity
Next is to set the number of heads (or tracks/cylinder). With 4 KB per track, 16 heads will gives a total of 64 KB per cylinder, which is convenient since the size of the backing file is 64 MB. Expert command (m for help): h
Number of heads (1-256): 16

Finally the number of cylinders will be set. It's important that the total size should match the actual size of the backing file. Since there are 64 KB per cylinder and 64 MB total, 1024 cylinders are needed. Expert command (m for help): c
Number of cylinders (1-131071): 1024
Now return to the normal menu (the "r" command): Expert command (m for help): r

E. Creating a Primary Partition

Create a new primary partition ("n" for new). Make it number 1. The defaults for the starting and ending cylinder are perfect because they will make the partition occupy the entire backing file, so press Enter when asked for the First and Last cylinder:
Command (m for help): n
Command action
Extended
Primary partition (1-4)
Partition number (1-4): 1
First cylinder (1-1024, default 1):
Using default value 1
Last cylinder or +size or +sizeM or +sizeK (1-1024, default 1024):
Using default value 1024
The new partition is created by default as a Linux partition [4]. Since you want to use the gadget with a Windows host, you should change the partition type (the "t" command) to FAT32 (code "b"): Command (m for help): t
Partition number (1-4): 1
Hex code (type L to list codes): b
Changed system type of partition 1 to b (Win95 FAT32)
Print out ("p") the new partition table to be sure everything's correct: Command (m for help):
Disk /root/data/backing_file: 16 heads, 8 sectors, 1024 cylinders
Units = cylinders of 128 * 512 bytes

Device Boot	***Start***	***End***	***Blocks***	***Id***	System
/root/data/backing_file1	1	1024	65532	b	Win95 FAT32

Finally write out ("w") the partition table to the backing storage:
Command (m for help): w

The partition table has been altered! Calling ioctl() to re-read partition table. Re-read table failed with error 25: Inappropriate ioctl for device. Reboot system to ensure the partition table is updated.
WARNING: If you have created or modified any DOS 6.x partitions, please see the fdisk manual page for additional information.

F. Adding a File System

At this point a new partition has been created but it doesn't yet contain a filesystem. The easiest way to add a filesystem is to load g_file_storage, connect the gadget to a USB host, and use the host to do the work. With a Linux host [13] run mkdosfs; with a Windows host. Or double-click on the drive's icon in the "My Computer" window.

G. Accessing the Backing Storage from the Gadget

It is possible to manipulate the data in the backing storage from the gadget (even to add the filesystem). Don't do this while the gadget is connected to a USB host! The key is to use the loop device driver with the "-o" (offset) option for the losetup program. For this to work, determine the partition's offset. Following the scheme given above would result in 4096. If not, one can use fdisk to find the correct offset value:
fdisk -lu /root/data/backing_file One must set cylinders.
This can also be done from the extra functions menu:
Disk data: 0 MB, 0 bytes
16 heads, 8 sectors/track, 0 cylinders, total 0 sectors

Units = sectors of 1 * 512 = 512 bytes

Device Boot	***Start***	***End***	***Blocks***	***Id***
System				
/root/data/backing_file1	8	8191	4092	b
Win95 FAT32				

Ignore the data at the top and concentrate on the table at the bottom. The required number is want is the value in the "Start" column. It gives the offset in sectors; to convert to bytes and multiply by 512. So we see that the offset is 8 x 512 = 4096 bytes.

Now use the losetup program to set up the loop device driver with the proper offset:

 # losetup -o 4096 /dev/loop0 /root/data/backing_file

Now /dev/loop0 is mapped to the partition within the backing storage. Create a file system on it: # mkdosfs /dev/loop0 and then one can mount it: *# mount -t vfat /dev/loop0 /mnt/loop*

Now you can transfer files back and forth. When this is done, make sure to unmount and detach the loop device: # umount /dev/loop0.

losetup -d /dev/loop0

H. Appearing as Virtual USB Device

Mounting the File system will let the mass storage device to appear virtually without attaching any physical hardware.

IV. CONCLUSIONS AND FUTURE ENHANCEMENTS

We have shown that virtual USB concept work well with mass storage class. This will prove itself a great testing environment as far as testing of USB drivers is concerned. Mass storage class was tested virtually using this framework and it proved to be a good working environment even when the actual device is under development phase. USB Host side drivers can be developed without the need of any real device. This approach can also help in testing mass storage devices of different capacities virtually.

- *Future Enhancements*

  Here are a few suggestions about the future enhancements:

  - *Implementing Other USB classes for the same Frame work*: Currently, the scope of project restricted to Mass storage class only. This can be extended to other USB classes. There are other USB classes like, Audio, Printer, Video, Communication classes etc. Using this frame work; similar to virtual mass storage; virtual printer, video, communication classes can be implemented.
  - *Exactly emulating any USB device:* This frame work can be extended to emulate any 'real' device. For example some one wants that virtual mass storage should behave exactly like Kingston 512MB device, this is also possible with the help of proposed framework. But definitely it requires more changes in existing framework to include this functionality.

REFERENCES

[1]. Compaq, Hewlett-Packard, Intel, Lucent, Microsoft, NEC, and Philips. Universal Serial Bus Specification - Revision 2.0, April 2000. http://www.usb.org.

[2]. Detlef Fliegl. Programming Guide for Linux USB Device Drivers - Revision 1.32, 2000. http://usb.cs.tum.edu/usbdoc

[3]. SystemSoft and Intel. Universal Serial Bus Common Class Specification – Revision 1.0, December 1997. http://www.usb.org/developers/devclass_docs.

[4]. Linux kernel sources. http://www.kernel.org/.

[5]. Linux USB sources http://www.linux-usb.org/

[6]. Intel. Universal Host Controller Interface (UHCI) Design Guide - Revision 1.1, March 1996. http://www.usb.org/developers/docs/

[7]. Intel. Enhanced Host Controller Interface for Universal Serial Bus - Revision 1.0, March 2002. http://www.usb.org/developers/docs/

[8]. Intel USB source http://www.intel.com/technology/usb/

[9]. Wikipedia USB source http://en.wikipedia.org/wiki/USB.

[10]. Wiki USB source, http://www.en.wikipedia.org/wiki/USB_flash_drive/

[11]. Linux USB source, www.linux-usb.org/USB-guide/x498.html

[12]. Mass Storage USB source www.gentoo-wiki.com/HOWTO_USB_Mass_Storage_Device

[13]. Linux USB source, www.linux.about.com/od/linux101/a/desktop04c.htm.

[14]. Linux USB sources, www.gsi.de/informationen/wti/library/scientificreport2006/PAPERS/INSTRUMENTS-METHODS-15.pdf

[15]. Linux USB source http://www.linux-usb.org/.