International Journal of Pure and Applied Mathematics
Volume 118 No. 22 2018, 1587-1594
ISSN: 1314-3395 (on-line version)
url: http://acadpubl.eu/hub
Special Issue
ijpam.eu

USB DEVICE DRIVER DEVELOPMENT

Priscilla Whitin[1], Smilee Mathuram[2], Vinoth John Prakash[3]
[1,3]Assistant Professor, [2]Associate Professor,
[1,2,3]Veltech Rangarajan Dr.Sagunthala R&D Institute Science & Technology, Chennai

Abstract: This project is about USB Device Driver Development. The project explores two platforms, Linux and Android for development of USB device drivers. The hardware and software tools available in the market to explore and understand USB were also studied.USB device drivers were developed for both Linux kernel-space and user space.

1. Introduction

USB, short for Universal Serial Bus, is an industry standard developed in the mid-1990s that defines the cables, connectors and communications protocols used in a bus for connection, communication, and power supply between computers and electronic devices. It is currently developed by the USB Implementers Forum (USB IF).

USB was designed to standardize the connection of computer peripherals (including keyboards, pointing devices, digital cameras, printers, portable media players, disk drives and network adapters) to personal computers, both to communicate and to supply electric power. USB has effectively replaced a variety of earlier interfaces, such as parallel ports, as well as separate power chargers for portable devices.

In computing, a device driver (commonly referred to simply as a driver) is a computer program that operates or controls a particular type of device that is attached to a computer. A driver provides a software interface to hardware devices, enabling operating systems and other computer programs to access hardware functions without needing to know precise details of the hardware being used.

A driver communicates with the device through the computer bus or communications subsystem to which the hardware connects. When a calling program invokes a routine in the driver, the driver issues commands to the device. Once the device sends data back to the driver, the driver may invoke routines in the original calling program. Drivers are hardware dependent and operating-system-specific. They usually provide the interrupt handling required for any necessary asynchronous time-dependent hardware interface.

1.1 Purpose

The main purpose of Device drivers is to provide abstraction by acting as translator between a hardware device and the applications or operating systems that use it. Programmers can write the higher-level application code independently of whatever specific hardware the end-user is using.

For example, a high-level application for interacting with a serial port may simply have two functions for "send data" and "receive data". At a lower level, a device driver implementing these functions would communicate to the particular serial port controller installed on a user's computer.

Kernel space can be accessed by user module only through the use of system calls. End user programs like the UNIX shell or other GUI- based applications are part of the user space. These applications interact with hardware through kernel supported functions.

1.2 Identifiers

A device on the PCI bus or USB is identified by two IDs which consist of 4 hexadecimal numbers each. The vendor ID identifies the vendor of the device. The device ID identifies a specific device from that manufacturer/vendor.

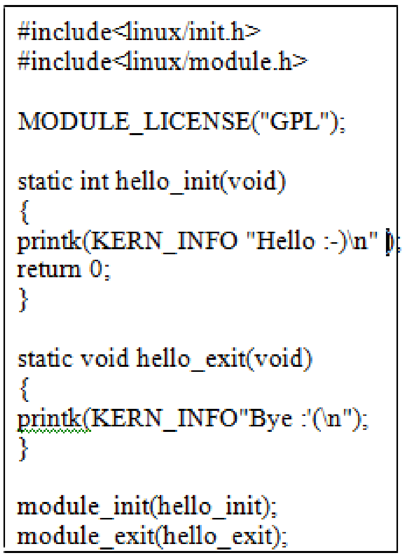
A PCI device has often an ID pair for the main chip of the device, and also a subsystem ID pair which identifies the vendor, which may be different from the chip manufacturer.

The main objectives of the project is to understand the USB 2.0 protocol and write USB device drivers to facilitate communication between user applications and the USB device. The platforms to be explored are Linux and Android.

2. Implementation & Results of USB Drivers in Kernel-Space

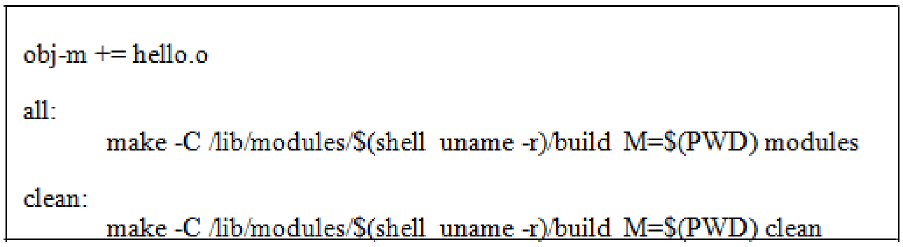
A simple prodecure to create,load and unload a kernel module is shown in Figure 2.1,Figure 2.2,Figure 2.3

and Figure 2.4.Create two files **hello.c** and **Makefile** and put them inside the same directory.



```
#include<linux/init.h>
#include<linux/module.h>

MODULE_LICENSE("GPL");

static int hello_init(void)
{
printk(KERN_INFO "Hello :-)\n" );
return 0;
}

static void hello_exit(void)
{
printk(KERN_INFO"Bye :'(\n");
}

module_init(hello_init);
module_exit(hello_exit);
```

Figure 2.1. 'hello.c' program for kernel-space

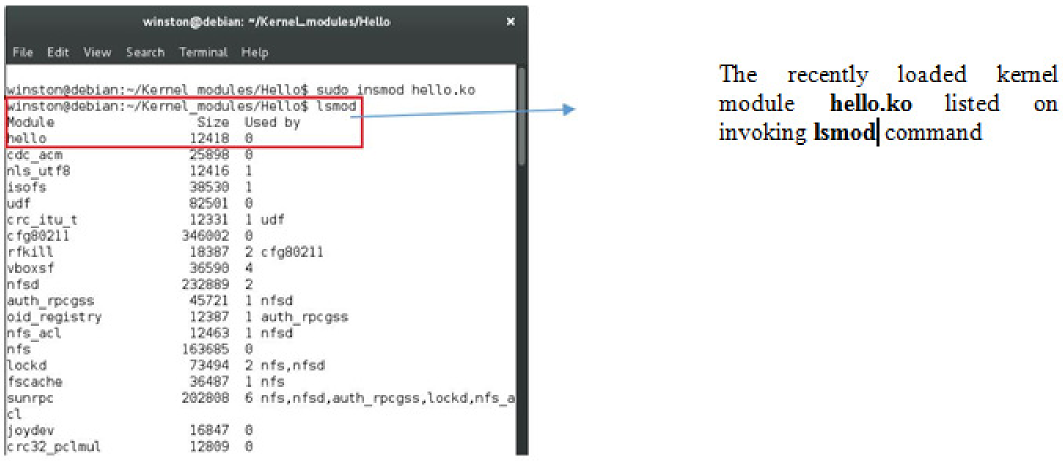


```
obj-m += hello.o

all:
        make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

clean:
        make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

Figure 2.2. 'Makefile' for building kernel module

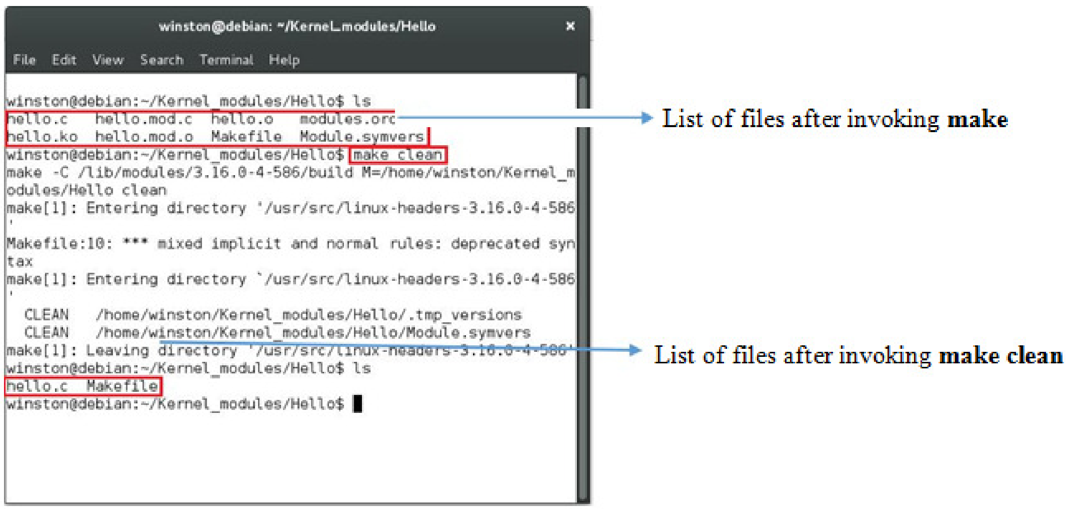
Navigate to the directory which contains **hello.c** and **Makefile**using terminal. If source file is **abcd.c** then the first line of **Makefile** should be **obj-m += abcd.o** .

- To build the kernel module from source file **hello.c,** use the command **make.**
- To clean-up the build, use **make clean**
- To load the kernel module, use **sudo insmod<module.ko>**
- To unload the kernel module, use **sudo rmmod<module.ko>**
- To list the kernel modules loaded , use **lsmod**
- To view printk driver messages , use **dmesg** or **dmesg –w**


winston@debian: ~/Kernel_modules/Hello
File Edit View Search Terminal Help
The recently loaded kernel module **hello.ko** listed on invoking **lsmod** command

```
winston@debian:~/Kernel_modules/Hello$ sudo insmod hello.ko
winston@debian:~/Kernel_modules/Hello$ lsmod
Module                  Size  Used by
hello                  12418  0
cdc_acm                25898  0
nls_utf8               12416  1
isofs                  38530  1
udf                    82501  0
crc_itu_t              12331  1 udf
cfg80211              346002  0
rfkill                 18387  2 cfg80211
vboxsf                 36590  4
nfsd                  232889  2
auth_rpcgss            45721  1 nfsd
oid_registry           12387  1 auth_rpcgss
nfs_acl                12463  1 nfsd
nfs                   163685  0
lockd                  73494  2 nfs,nfsd
fscache                36487  1 nfs
sunrpc                202808  6 nfs,nfsd,auth_rpcgss,lockd,nfs_a
cl
joydev                 16847  0
crc32_pclmul           12809  0
```

Figure 2.3. lsmod" command output


winston@debian: ~/Kernel_modules/Hello
File Edit View Search Terminal Help
List of files after invoking **make**
List of files after invoking **make clean**

```
winston@debian:~/Kernel_modules/Hello$ ls
hello.c   hello.mod.c  hello.o   modules.orc
hello.ko  hello.mod.o  Makefile  Module.symvers
winston@debian:~/Kernel_modules/Hello$ make clean
make -C /lib/modules/3.16.0-4-586/build M=/home/winston/Kernel_m
odules/Hello clean
make[1]: Entering directory '/usr/src/linux-headers-3.16.0-4-586
'
Makefile:10: *** mixed implicit and normal rules: deprecated syn
tax
make[1]: Entering directory `/usr/src/linux-headers-3.16.0-4-586
'
  CLEAN   /home/winston/Kernel_modules/Hello/.tmp_versions
  CLEAN   /home/winston/Kernel_modules/Hello/Module.symvers
make[1]: Leaving directory '/usr/src/linux-headers-3.16.0-4-586'
winston@debian:~/Kernel_modules/Hello$ ls
hello.c  Makefile
winston@debian:~/Kernel_modules/Hello$ 
```

Figure 2.4. make clean" command output

A blueprint for writing usb drivers is available at **<KERNEL_SRC>/drivers/usb/usb-skeleton.c**[1].The usb-skeleton driver is for devices with bulk end-points.

The **usb-skeleton.c**[1] was modified to make it work with our microcontroller loaded with usb_dev_bulk example.

3. Implementation & Results of USB Drivers in User-Space

USB drivers can also be implemented in user space. One popular solution is "libusb" from http://www.libusb.org/ .

It can also be used as a prototyping tool before writing kernel space USB drivers.[3]

Blinking the Keyboard Leds (Control Transfer) [6]

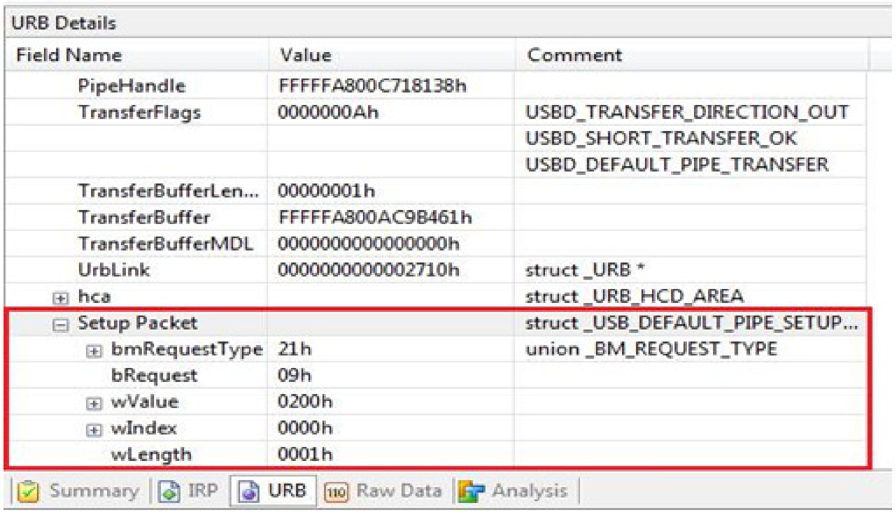
Keyboard LEDs works as follows

 User presses a key such as CAPS LOCK, NUM LOCK or SCROLL LOCK.

 Keyboard makes an interrupt transfer to the USB host(PC) by sending the keycodes (refer Figure 3.2)

 OS requests the USB host to initiate a control transfer with a setup packet of the attributes **bmRequest=0x21 bRequest=0x09 wValue=0x0200 wIndex=0x0000 wLength=0x0001** as shown in Figure 3.1

Data - 0x00 to 0x07 (Refer Table 3.1)



URB Details

Field Name	Value	Comment
PipeHandle	FFFFFA800C718138h	
TransferFlags	0000000Ah	USBD_TRANSFER_DIRECTION_OUT
		USBD_SHORT_TRANSFER_OK
		USBD_DEFAULT_PIPE_TRANSFER
TransferBufferLen...	00000001h	
TransferBuffer	FFFFFA800AC9B461h	
TransferBufferMDL	0000000000000000h	
UrbLink	0000000000002710h	struct _URB *
hca		struct _URB_HCD_AREA
Setup Packet		struct _USB_DEFAULT_PIPE_SETUP...
bmRequestType	21h	union _BM_REQUEST_TYPE
bRequest	09h	
wValue	0200h	
wIndex	0000h	
wLength	0001h	

Figure 3.1. Setup Packet to control keyboard LEDs

Table 3.1. Data and status of keyboard LEDs

DATA	SCROLL LOCK	CAPS LOCK	NUM LOCK
0x00	0	0	0
0x01	0	0	1
0x02	0	1	0
0x03	0	1	1
0x04	1	0	0
0x05	1	0	1
0x06	1	1	0
0x07	1	1	1

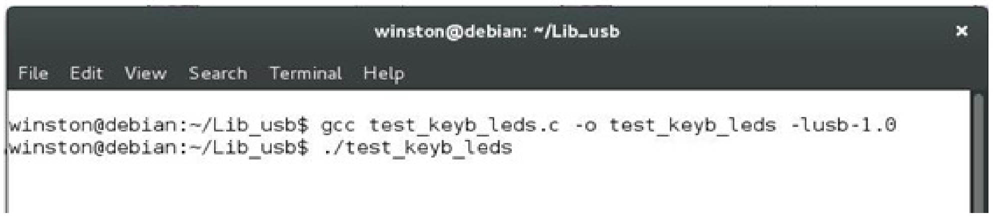
1-ON
0-OFF

3.2.1 LibUSB Program to test Keyboard LEDs

The below program in Figure 3.3 tests the keyboard LEDs in all possible combinations. The Table 3.1 shows the data transmitted and keyboard. Data from 0x00 to 0x07 is sent via control transfer.
The user-space program is shown in Figure 3.3 and compilation commands are specified in Figure 3.4 .

```
#include<libusb-1.0/libusb.h>
#include<stdlib.h>
#include<stdio.h>
#include<time.h>

int main(void)
{
libusb_device_handle *handle1=NULL;
time_t t = 0;
unsigned char data=0,i=0;

//Initialize libusb
libusb_init(NULL);

//Dell Entry Level Keyboard ---> Vendor ID =0x413C, Product ID =0x2107
handle1=libusb_open_device_with_vid_pid(NULL,0x413C,0x2107);

//Detach kernel driver from interface 0
libusb_detach_kernel_driver(handle1,0);

while(1)
{

for(i=0;i<=7;i++)                               1,
{                                               0); /*bmRequestType /*
data =i;                                        /*bRequest*/
libusb_control_transfer(handle1,                /*wValue */
0x21,                                           /*wIndex*/
0x09,                                           /*Ptr to data*/
0x200,                                          /*Length in bytes*/
0,                                              /*Timeout */
&data,

            t=time(NULL)+1;
            while((time(NULL)-t)<0);
            if(data==7) break;
            }
            if(data==7) break;
    }
//Release the USB device
libusb_close(handle1);
return 0;
}
```

Figure 3.3. Userspace program 'test_keyb_leds.c' to test USB keyboard LEDs


winston@debian: ~/Lib_usb
File Edit View Search Terminal Help

```
winston@debian:~/Lib_usb$ gcc test_keyb_leds.c -o test_keyb_leds -lusb-1.0
winston@debian:~/Lib_usb$ ./test_keyb_leds
```

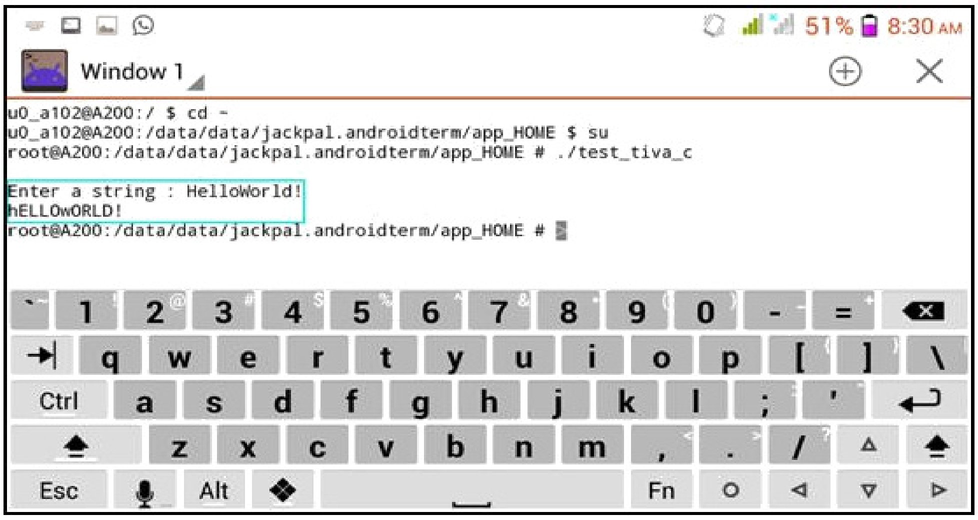
3.3 Rewriting Kernel Bulk Transfer Driver for User-Space

Refer to Section 2.2 for Kernel Bulk transfer driver. The below program shown in Figur 3.5 emulates the behavior of the kernel driver in user-space.

The compilation process and output is shown in Figure 3.6

```
#include<libusb-1.0/libusb.h>
#include<stdio.h>
#include<string.h>
int main(void)
{

libusb_device_handle *handle1=NULL;
int transferred=0;
unsigned char data[64]={};

//Initialize libusb

libusb_init(NULL);

//TI Generic Bulk Device ---> Vendor ID =0x1cbe,
Product ID =0x0003 handle1=libusb_open_device_with_vid_pid(NULL,0x1cbe,0x0003);

//Get string from user
printf("\nEnter a string : ");
scanf("%s",data);

//Write the string to bulk OUT endpoint 0x01

libusb_bulk_transfer(handle1,0x01,data,64,&transferred,0);

//Read the string to bulk IN endpoint 0x81

libusb_bulk_transfer(handle1,0x81,data,64,&transferred,0);

//Print the string retrieved from the device printf("%s\n",data);

//Release the USB device
libusb_close(handle1);

return 0;
}
```

Figure 3.5. User-space program 'test_tiva_c.c' to test bulk transfer



```
u0_a102@A200:/ $ cd ~
u0_a102@A200:/data/data/jackpal.androidterm/app_HOME $ su
root@A200:/data/data/jackpal.androidterm/app_HOME # ./test_tiva_c

Enter a string : HelloWorld!
hELLOwORLD!
root@A200:/data/data/jackpal.androidterm/app_HOME #
```

Figure 3.8. Running test_tiva_c program for bulk transfer with Tiva C Launchpad in Android

4. Conclusion

Thus an introductory level work was carried out in USB kernel-space and user-space device drivers for Android and Linux.More work has to be carried out in the area USB firmware for microcontrollers.Other platforms such as Windows would also be explored for the scope of device driver development .

Implementing protocol converters such as USB to CAN ,USB to LIN etc., and writing drivers for them would be carried out in the future.

References

[1] “The Linux Kernel Archives” at http://www.kernel.org

[2] “Linux USB”, www.linux-usb.org

[3] "LibUSB project", http://www.libusb.org/

[4] Greg Kroah-Hartman,”Linux kernel in a Nutshell”, O’Reilly &Associates,Dec. 2006

[5] Jonathan Corbet,Alessandro Rubini and Greg Kroah-Hartman," Linux Device Drivers" ,O’Reilly &Associates",Jan. 2005

[6] “USB-IF”, www.usb.org

[7] www.wikipedia.org