International Journal of Advanced Research in Electronics and Communication Engineering (IJARECE)
Volume 4, Issue 4, April 2015

Design Of Linux USB Device Driver For LPC2148 Based Data Acquisition System Including GSM.

Snehal A. More, Tejashree R. Padwale, Anuja B. Sapkal, Prof. Pradeep R. Taware

***Abstract*- Among several other advantages of free operating systems like Linux, is which the internals are generally open for all view. Device drivers take on a special role in the Linux kernel. Writing device drivers has always been a tedious job. This paper presents useful tools for developing USB device drivers under Open SUSE Linux OS. Data acquisition is simply the process of bringing a real-world signal, into the computer, for processing, storage, analysis or other data manipulation. Analog data is generally acquired and transformed into the digital form for the purpose of processing, transmission and display. This project is to simulate real-world data acquisition system using LPC2148.Which is having ARM7TDMI-S core and on-board USB interface for communicating with Host PC system.LPC2148 will have customized USB stack hosted which will acquire data from various sensors connected to ADC channels and sends acquired data using USB interface to Master and GSM through mobile phone.**

***Keywords*:-Linux, Device driver, USB device, Data acquisition system.**

I. INTRODUCTION

The universal serial bus (USB) is a connection between a host computer and a number of peripheral devices. The latest versions of the USB specification added high-speed connections with a theoretical speed limit. Topologically, USB subsystem just isn't laid out as a shuttle bus, it is rather a tree built out of several point-to-point links. The links are four-wire cables which are power, ground and two signal wires, that connect a device and a hub. At the technological level bus is inelastic, as it is a single master implementation in which the host computer polls the numerous devices. Despite this intrinsic limitation, the bus has many interesting characteristics, such as the ability for a device to request a preset bandwidth for its data transfers so as to reliably support video and audio I/O. Another important feature of USB is which it acts merely as a communication channel between device and the host, without requiring actual meaning or structure for the data it delivers. Device driver is a program that links a peripheral gadget to OS of computer and hide completely the important points of how the device works.[2]

II. USB DEVICE DRIVER

The particular USB protocol specifications define some standards that any device of the specific type can follow. When a device follows that standard, then the special driver for that device is not necessary. These distinct types are called classes and include things like things like storage devices, mice, key-boards, network devices, in addition to modems. Other types of devices that do not fit into these classes call for a special vendor-specific driver to be written with the specific device. USB to serial devices and Video devices are demonstration where there is no defined standard, and a driver is used for every different device coming from different manufacturers.

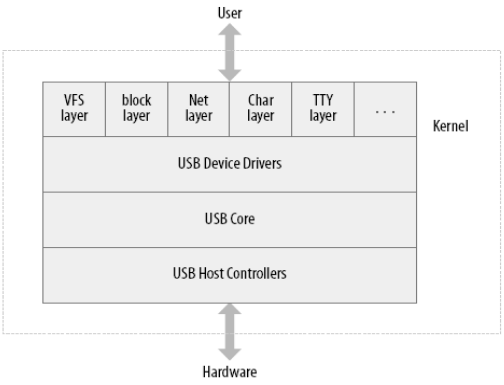

Fig.1 USB driver overview

USB drivers live between the different kernel subsystems (block, net, char, etc.) and the USB hardware controllers. The USB core feeds an interface for USB drivers to use to access and control the USB hardware, without different kinds of USB hardware controllers that are present on the system.[8]

III. USB DEVICE BASICS

Endpoints

The most basic form of USB communication will be through something called an endpoint. A USB endpoint can carry data in barely one direction, either from the host computer on the device (OUT endpoint) or on the device to the host personal computer (IN endpoint). Endpoints are unidirectional pipes. A USB endpoint can be one of four different types that describe how the data is transmitted:

Control

Control endpoints are used to allow access to various areas of the USB device. They may be used for configuring the system, retrieving information about the system, sending commands to the system, or recovering status reports regarding to the unit.

Interrupt

Interrupt endpoints transfer a small amount of data at a fixed rate each and every time the USB host asks the product for data. They are also broadly used to send data to USB devices to discipline the device, but are not generally transfer large amounts of data.

Bulk

ISSN: 2278 – 909X

Bulk endpoints transfer large amounts of data. Bulk endpoints are much larger than interrupt endpoints.These transfers are not guaranteed through the USB protocol to always allow it to become through in a specific time frame.
Isochronous
Isochronous endpoints also transfer remarkable amounts of data, but the transfer of data is not always guaranteed. These endpoints in the devices that can handle lack of data, and depend more on keeping consistent stream of data flowing.[8]

Interfaces

USB endpoints are bundled up into interfaces. USB interfaces handle one type of a USB rational connection, such as a sensitive a keyboard, mouse, or audio mode. USB interfaces may have change settings. The first state of a interface is in the initial setting, numbered 0. Alternate settings may be used to control individual endpoints in different methods, such as to reserve different levels of USB bandwidth for the gadget. USB interfaces are described inside the kernel with the struct usb_interface framework. This structure is what the USB core passes to USB sdrivers which is what the USB driver then manages controlling. If the USB motorist certain to this particular user interface utilizes the particular USB important quantity which is major number, this kind of variable provides the minor number given through the USB core towards the user interface. This is valid only after a successful contact to usb_register_dev There are other fields in the struct usb_interface structure, however USB individuals don't need to know about all of them.[2]

Configurations

USB interfaces are themselves bundled way up into configurations. A USB device can include multiple configurations and might switch between them as a way to change the state of the unit. Linux describes USB configurations while using structure struct usb_host_config and entire USB devices while using structure struct usb_device. A USB device driver commonly should convert data from a given struct usb_interface structure into a struct usb_device structure that your USB core needs for a variety of function calls.

Descriptors

All USB devices have a descriptors which describe the host information such as device, Owner of device, supported version, configuration ways, endpoint number and types etc. The more common USB descriptors are Device Descriptors, Interface Descriptors, Configuration Descriptors, Endpoint Descriptors.
USB devices can only have one device descriptor. The device descriptor of USB device contains information about, the Product and Vendor IDs used, maximum packet size and the number of configurations of the device. The configuration descriptor shows how the USB device is powered, the maximum power consumption, and the number of interfaces.[]The particular user interface descriptor may be seen as an header as well as grouping on the endpoints right into a sensible collection performing just one element on the device. Endpoint descriptors utilized to explain endpoints other than endpoint no.

IV. REQUESTS

Request related to descriptors in device driver are given for descriptor handlers it is MAX_DESC_HANDLERS. For device descriptor field offsets DESC_bLength and DESC_bDescriptorType. For configuration descriptor field offsets CONF_DESC_wTotalLength, CONF_DESC_bConfigurationValue, CONF_DESC_bmAttributes. For interface descriptor field offsets it is INTF_DESC_bAlternateSetting and for endpoint descriptor field offsets ENDP_DESC_bEndpointAddress and ENDP_DESC_wMaxPacketSize.[4]

abStdReqData-This is used for data storage area for standard requests.

HandleUsbReset()-This is used for USB reset handler.

USBHwInit()-Initialize hardware by calling function USBHwInit().

USBHwRegisterDevIntHandler()-Register bus reset handler by calling this function.

USBHwRegisterEPIntHandler()-Register control transfer handler on EP0 i.e. Logical Enpoint 0.

USBHwEPConfig()-Setup control endpoints i.e.Logical Enpoint 0.

USBRegisterRequestHandler -Register standard request handler by sending this request.

USBRegisterDescriptors()-Registers a pointer to a descriptor block containing all descriptors for the device.

USBGetDescriptor()-Parses the list of installed USB descriptors and attempts to find the specified USB descriptor.

USBSetConfiguration()-Configures the device according to the specified configuration index and alternate setting by parsing the installed USB descriptor list.

HandleStdInterfaceReq()-Local function to handle a standard interface request.

HandleStdDeviceReq()-Local function to handle a standard device request.

HandleStdEndPointReq()-Local function to handle a standard endpoint request.

USBHandleStandardRequest()-Default handler for standard requests.

USBRegisterCustomReqHandler()-Registers a callback for custom device requests.

There are some standard requests which are used while writing USB device driver are such as,

REQ_GET_STATUS

REQ_SET_ADDRESS

ISSN: 2278 – 909X

REQ_GET_DESCRIPTOR

REQ_GET_CONFIGURATION

REQ_SET_CONFIGURATION

REQ_CLEAR_FEATURE

REQ_SET_FEATURE

REQ_SET_DESCRIPTOR

REQTYPE_RECIP_DEVICE

REQTYPE_RECIP_INTERFACE

REQTYPE_RECIP_ENDPOINT

V. DESIGN IMPLEMENTATION

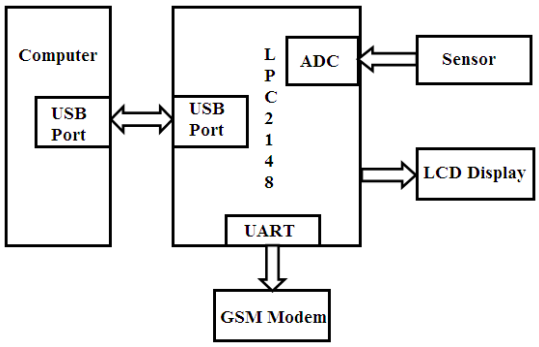

Fig.2 Block Diagram of Data acquisition system

Data acquisition system is only as an application of device driver.In data acquisition system connect the sensor to inbuilt ADC of LPC2148 microcontroller.Measure and display the values of sensor on LCD display.Measured data is transfer to computer with the help of USB device.Connect one end of USB device to USB port of LPC2148(Slave) and another end to USB port of PC system (Master),that computer does not support the transmission then load device driver and send data to computer.We can also send collected data to any mobile through the GSM modem.[9]

VI. USB COMMUNICATION FLOW

The USB includes a communication flow involving computer software about the coordinator and its USB function. Capabilities will get various communication move desires for various client-to-function will be. The USB provides greater overall bus utilization by allowing the various communication passes in order to USB function. Every single communication flow works by using some bus access to perform communication involving client and function. Each communication flow will be terminated at an endpoint more than a device. Device endpoints are widely-used to distinguish aspects of communication flow.[1]

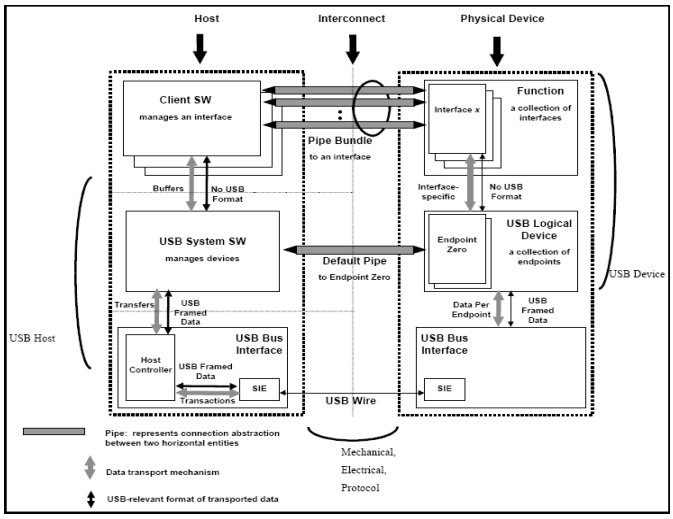

Fig.3 USB Host/Device Detailed View

USB Physical Device: A piece of computer hardware on the end of a USB cable that performs some useful end user function.
Client Software: The software which process on the host, corresponding to a USB device. This client software is usually supplied with the operating system or provided along with the USB device.
USB System Software: The software that supports the USB in a particular operating system. The USB System Software is usually supplied with the operating system, independently with particular USB devices or client software.
USB Host Controller (Host Side Bus Interface): The hardware and software that enables USB devices to be attached to a host.
Host Controller Driver (HCD): The program software interface between the Hardware Host Controller and hardware System Software. One USB Driver may support different Host Controllers without requiring specific knowledge of a Host Controller rendering. A Host Controller implementer provides an HCD implementation that sustains the Host Controller.
USB Driver (USBD): The actual software in relation to this USB program computer software as well as the client computer software. This software provides clients with effortless functions for manipulating USB devices. A USB logical device seems to the USB system as an amount of endpoints. Endpoints are grouped into an interface. Interfaces are views towards function. The USB System Software manages the device using the Default Manage Pipe. Client software manages the interface using pipe bundles. Client software requests which data be moved over the USB between a buffer for the host and an endpoint for the USB device. The Sponsor Controller (or USB device, depending on transfer direction) packetizes the information to move it above the USB. The Host Controller additionally coordinates when bus access is needed to move the packet of data above the USB.[7]

VII. USB DATA FORMAT

USB data is sent in packets Least Significant Bit (LSB) first. The actual packets are usually next enclosed directly into

ISSN: 2278 – 909X

support frames to make a USB message. There are four main USB packet types are Token, Data, Handshake and Start of Frame. Every packet can be constructed from diverse field types, namely SYNC, PID, Address, Data, Endpoint, CRC and EOP. Packets block of information with a defined data structure. The packet is the lowest level of the USB transfer hierarchy describing the physical layer of the interface. If you were to monitor D+ and D- you would see the packet fields such as Packet identifier, Address, Endpoint, Data, Frame number, CRC. The actual PID signals for the receiver which just what packet framework along with information are going to be along with the way the receiver must act in response.

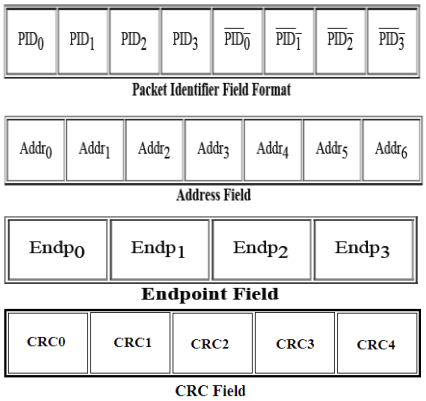

Fig.4 Data Format

VIII. ENUMERATION

The USB specification defines six device states. During enumeration steps, a computer device techniques through four on the claims which are Powered, Default, Address, and Configured. The other two claims are usually Attached and Suspend. Within each state, it has defined features and behavior.

1. The person attaches a USB device to a USB port or the device powers up which has device already connected. The actual hub provides power to that port, as well as the device is within the Powered state.
2. The hub detects the USB device at port. The hub monitors the voltages inside the transmission lines coming from all of its places. Whenever a device plugs right into a port, device's pull-up offers its line high voltage, enabling the hub to detect that a device is attached to the USB port. On detecting a device, the hub continues to deliver power but doesn't still transmit USB traffic on the device.
3. The actual host learns about the completely new device. Every hub operates by having its interrupt endpoint in order to report activities from the hub. About understanding of functionality, the host sends request to a new hub Get_Port_Status for more information. The information went back tells the host every time a device is recently attached.
4. The particular hub detects whether one device is low in addition to full speed. The hub determines whether the device is low or full speed by examining the voltages within the two signal outlines. The hub covers the speed associated with device by finding out which line provides the higher voltage at any time idle. The actual hub transmits the details for the host in reply to an increased Get_Port_Status obtain.
5. The hub resets the device. Whenever a host learns connected with the completely new device, host controller communicates the hub a new Set_Port_Feature demand which questions hub as a way to reset the port. The hub sends the reset simply to the new device. Other hubs and devices across the bus don't see the reset to zero.
6. The actual hub ensures an indication path involving the unit device the bus. After reset, the USB device is in the Default state of enumeration. The device's USB registers are working their reset states as well as the device can respond to command transfers from Endpoint 0. The device convey with the host using the default address of 00h.
7. The actual host transmits some sort of Get_Descriptor request to discover the maximum packet size. The specific host sends the request to USB device with address 0, Endpoint 0. Because the host enumerates only one device at a time, even when several devices attach immediately.
8. The host assigns an address. The host controller assigns an original address to the device by simply sending a Set_Address request. It completes the Status stage on the request using the default address and implements the new address. It is now in the Address state. All Communications using this aspect on utilize the new address.
9. The host learns about the device's abilities. The actual host sends a Get_Descriptor request towards completely new address to read the device descriptor. This time the host retrieves the entire descriptor of the device. The descriptor is a data structure containing the maximum packet size for Endpoint 0, the number of configurations that device supports, and other basic information about the device attached.
10. The host assigns and loads a device driver for a particular device. After learning about the device from its descriptors, the host looks to find the best match in a device driver to manage communications using the device. If there is no match,it looks for the match having any class, subclass, and protocol values retrieved from the device.
11. The host's device driver selects a configuration. After learning with regards to a device from the descriptors, the device driver requests a setting by sending some sort of Set_Configuration request using the desired configuration variety.

The other two device states are Attached and Suspend. If the hub isn't providing power to a VBUS line, the device is in the Attached state. A device enters the Suspend state after detecting no bus activity. Each configured and unconfigured units must support this state.[5]

Device Removal

When a user removes a USB device from the port, the hub disables the device's port and host learns that the removal take place, learning that an event has occurred, and sending a Get_Port_Status request to find the event.

IX. RESULT

We have successfully install USB device driver.

ISSN: 2278 – 909X

```
linux:/home/teju/usb_lastfinal/usb_driver # make i
insmod usb_driver.ko
lsmod | head
Module                  Size  Used by
usb_driver              4956  0
ip6t_LOG                5150  6
xt_tcpudp               2107  2
xt_pkttype               912  3
xt_physdev              1539  2
ipt_LOG                 5119  6
xt_limit                1705  12
rfcomm                 69525  4
sco                    16711  2
dmesg | tail -20
[  176.247124] usb 2-1.1: Product: USB ARM
[  176.247128] usb 2-1.1: Manufacturer: SDSN Inc.
[  176.247131] usb 2-1.1: SerialNumber: 00000001
[  179.369636] my_usb_driver_init: Registering my usb driver to USB subsystem
[  179.369656] my_probe: succesfull memory allocation for usb_dev
[  179.369659] my_probe: no. of endpoints = 3
[  179.369712]
[  179.369714] my_probe: bulk in 1 endpoint addr  = 0x85
[  179.369716] my_probe: Max Packet Size = 64
[  179.369718] my_probe: bulk out 1 endpoint addr = 0x8
[  179.369720] my_probe: Max Packet Size = 64
[  179.369721]
[  179.369723] my_probe: USB device attached to my_usb_driver minor = 222
[  179.369725] my_probe: new full speed usb device found
[  179.369727] my_probe: idVendor=ffff, idProduct=0004
[  179.369729] my_probe: Product: USB ARM
[  179.369731] my_probe: Manufacturer: SDSN Inc.
[  179.369733] my_probe: SerialNumber: 00000001
[  179.369734] my_probe: successful.
[  179.369752] usbcore: registered new interface driver my_usb_driver
linux:/home/teju/usb_lastfinal/usb_driver #
```

We have successfully remove USB device driver.

```
linux:/home/teju/usb_lastfinal/usb_driver # make r
rmmod usb_driver.ko
lsmod | head
Module                  Size  Used by
ip6t_LOG                5150  6
xt_tcpudp               2107  2
xt_pkttype               912  3
xt_physdev              1539  2
ipt_LOG                 5119  6
xt_limit                1705  12
rfcomm                 69525  4
sco                    16711  2
bridge                 71636  1
dmesg | tail -20
[  220.334693] usb 2-1.1: Manufacturer: SDSN Inc.
[  220.334697] usb 2-1.1: SerialNumber: 00000001
[  220.335595] my_probe: succesfull memory allocation for usb_dev
[  220.335602] my_probe: no. of endpoints = 3
[  220.335708]
[  220.335713] my_probe: bulk in 1 endpoint addr  = 0x85
[  220.335719] my_probe: Max Packet Size = 64
[  220.335724] my_probe: bulk out 1 endpoint addr = 0x8
[  220.335728] my_probe: Max Packet Size = 64
[  220.335732]
[  220.335736] my_probe: USB device attached to my_usb_driver minor = 222
[  220.335741] my_probe: new full speed usb device found
[  220.335747] my_probe: idVendor=ffff, idProduct=0004
[  220.335752] my_probe: Product: USB ARM
[  220.335756] my_probe: Manufacturer: SDSN Inc.
[  220.335762] my_probe: SerialNumber: 00000001
[  220.335766] my_probe: successful.
[  226.763530] my_usb_driver_exit: Removing my usb driver from USB subsystem
[  226.763541] usbcore: deregistering interface driver my_usb_driver
[  226.763668] my_desconnect: USB device 222 now disconnected
linux:/home/teju/usb_lastfinal/usb_driver #
```

I. CONCLUSION

We have successfully designed USB device driver for LPC2148 using Linux operating system and Data acquisition system.This system can read the sensor values and send it to host for storing purpose.Using GSM modem we can also send data to any mobile or remote host.

REFERENCES

[1] An Embedded USB controller linux device driver by Dimitris Lampridis.

[2] Linux Device Drivers 3rd Edition by Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman

[3] http://ecos.sourceware.org/docs3.0/ref/usbsenum.html

[4] Programming Guide for Linux USB Device Drivers by Detlef Fliegl,

[5] Useful USB Gadgets on Linux February, 2012 by Gary Bisson,Adeneo Embedde

[6] Writing device drivers in Linux: A brief tutorial by Xavier Calbet

[7] Writing USB Device Drivers by Greg Kroah-Hartman

[8] "Universal Serial Bus Revision 2.0 specification". Universal Serial Bus Micro-USB Cables and Connectors Specification. USB Implementers Forum, Inc. 4 April 2009.

[9] LPC2148 USB Mass Storage Device Example http://www.keil.com/download/docs/307.asp

ISSN: 2278 – 909X