USB INTERFACE AND DRIVER
Mentor: Dr. Yann Hang Lee
Team Members: Jubin Mehta, Koshik Samota
Email: (jmehta3, ksamota)@asu.edu

INTRODUCTION

With so much advancement in technology, producing, processing and sharing, information on the go is possible. The technological wonders as the Tablets, Notebooks and Smartphones have one common feature – ‘Mobility’. No one likes to carry around things that are heavy. Over the years, there has been tremendous size reduction in all electronic devices. The CPU’s of the old desktop PC’s used to be very bulky with a lot of wired connections for the various external devices to be attached to the computer. We can’t have such bulky machines anymore as it would violate the whole idea of producing information ‘On the Go’. Let us consider the older machine. For a printer to be connected to a CPU, we would connect it either to a serial or a parallel port. What we did not get was efficient performance out of it. Input devices like the mice and keyboard had a different port for their connection increasing the wiring. Other devices like a PDA or a digital camera were connected to serial bus which was very low on performance in terms of speed. Let us identify the complexity of the old system. Firstly, each device needed an individual protocol for its connection, making the system wiring very complex. Secondly, the performance suffered due to low speeds provided for file transfer between the two devices. Thirdly, there was always a restriction on the number of devices that could be connected to your computer at a given instance of time.Fourthly, the hardware for all the above mentioned protocols was very expensive.

It was in 1994, when seven companies tried to come up with a solution to solve out the complexity. This was the point when USB as a protocol came into existence wiping out the problems existing in the previous system. USB provided a common architectural protocol for a number of devices to be connected to a host machine reducing the complexity on both software and hardware fronts. The improved versions of the USB protocol provided speeds in the range of 12 Mbps to 480 Mbps improving performance levels for file exchange. USB root hub supports 127 connections at a given time to a system using sub hubs and bridges. Also, the cost for implementing USB on both Hardware and Software fronts is very minimal, not to forget the ease with which it is used. The simplicity of the protocol and reducing load on both hardware and software front assisted USB as a protocol to come into prominence and extensive use.

EVOLUTION OF USB

The design goals of a new standard should overcome the existing shortcomings of the previous standards as perceived by manufacturers and users, while providing for further growth, performance, and expansion.

USB 1.0: The ground breaking technology was released as USB 1.0 specification which supported low speed data transfer of about 1.5 Mbps – 12 Mbps. This type of speed was supported by interactive devices such as keyboards, mouse and stylus (3).

USB 1.1: The second release of USB was the 1.1 specification which also supported low speed data transfer as the previous standard, but overcome some of the problems related to the earlier standard like problems associated with the hubs (3).

USB 2.0: The most popular version of the USB protocol used is the USB 2.0 specification.
This specification supports High Speed Data Transfer speed of about 480 Mbps. This transfer type was supported by applications which involved transfer of video and audio data and also data to and from USB storage devices as pen-drives and external storage devices (3).

**USB 3.0:** The latest release of the USB is the 3.0 specification which supports ‘Super-speed’ data transfers of speeds around 5 Gbps which can be used for heavy data transfers with time constraints.

**BUS TOPOLOGY**

The tiered star topology is used by the USB devices and hubs to get connected to the Root hub in the host as shown in the diagram. The tiered star topology is maintained to avoid circular connections.

![Figure 1: Bus Topology (3)](image)

The USB attachments are made by connecting through a HUB. At the root of the tree is the Root Hub which has the pre-defined address of 0. It can be further connect to other hub or a function (device). Each of this connection is a wired, point to point connection. The propagation delay suffered by the data to flow from the host to the devices through the cables restricts the number of tiers in the USB topology to 7(3). Also since the addressable units are of 8 bits, hence, only 127 devices can be connected. The hubs can occupy layers except the first and seventh. The last tier supports connection only to devices. A compound device occupies 2 tiers, hence cannot be attached at tier 7(3). An example of a compound device is a keyboard with a trackball. Both of these are treated as a single functioning device but are connected to single common hub. For different functionalities of the device, a separate address is allocated to a particular functionality.

Although a device might be connected physically to a hub in tier 5, but it is logically treated to be connected directly to the root hub.

**Host Controllers in USB**

A Host Controller IN USB is a register level interface that allows the USB Hardware to communicate with the Host Controller Driver which is already provided by the Operating System. All controllers conform to either of the following STANDARDS.

- Universal Host Controller Interface (UHCI)
- Open Host Controller Interface (OHCI)
- Enhanced Host Controller Interface (EHCI)
- USB OTG Controllers
- eXtensible Host Controller Interface

1. **Universal Host Controller Interface (UHCI):** This Interface was created by Intel to support Low Speed and Full Speed Data Transfer Speeds (USB 1.0). Most of the Intel based personal computers have this kind of Interface (8)(4).

2. **Open Host Controller Interface (OHCI):** It was developed by companies like Microsoft and Compaq. This Interface supports Low and Full Data Transfer Speeds (USB 1.1). The OHCI compatible Host Controller compatible Hardware chips have much more intelligent build into them then the UHCI compatible chips and hence more efficient (8)(4).
3. **Enhanced Host Controller Interface (EHCI):** This Interface supports High Speed Data Transfer (USB 2.0). It has attached along a ‘Companion Controller’, either an Open Host Controller Interface or a Universal Host Controller Interface to support the lower speed devices. Hence, whenever a device supporting slower speed is attached to the Host, then the EHCI transfers control to the Companion Controller. It is possible to have all above mention Controller Interface co-exist on a single system (8)(4).

4. **USB OTG Controller:** A new feature which is gaining a lot of popularity is the OTG. By this, a device is given a capability to assume DUAL–ROLES. It can either configure itself as a host or a device (4).

5. **eXtensible Host Controller Interface:** The latest Interface Standard is the eXtensible Host Controller Interface. It is designed to replace all the earlier versions of the Interface. It supports USB 3.0 specifications and is Backward Compatible with USB 1.0, 1.1 and 2.0 devices (6).

**USB Backward Compatibility:**

The backward compatibility feature of USB let us connect Lower speed Devices to Host Controllers which support higher speeds of the newer specifications. It is either done using ‘Companion Controller’ as in EHCI or making a provision for all data speeds as in eXtensible Host Controller Interface.

**ADDRESSING IN USB**

**Endpoints:**

An endpoint provides an identity to the USB device portion where the communication takes place between the host and the device. An endpoint is not a physical port on the USB devices but a logical port on the device end. There are many virtual endpoints in the USB device which do not dependent on each other. When the device is attached to the host, the logical device is assigned a unique address number from 1 to 127. Also, when a device is designed, it is provided with a unique device-determined identifier called the endpoint number. The direction in which the data will flow through the endpoints is decided by the USB device. Thus, in order to uniquely address an endpoint we need a combination of the unique device address (assigned by the host), the endpoint number and the flow direction. Thus, depending on the connection of the endpoint and the direction in which the data flows, the endpoints are referred as:

- IN (input) endpoint (from the device to the host)
- OUT (output) endpoint (from the host to the device)

Also, every endpoint is associated to one of the data transfer type. Thus, depending on the data transfer the endpoint is related to they are again classified into control endpoint, bulk endpoint, isochronous endpoint and interrupt endpoint. Thus, we can have either a bulk IN endpoint or a bulk OUT endpoint.
The endpoint 0 (zero) is of vital importance for the default control method which all the USB devices are required to implement. The host uses this method for the initialization and manipulation of the logical USB devices. The configuration details of the USB device can be obtained from the logical pipes connected to the input and the output endpoint 0. This pipe can also be used to have an access to the USB control and status information. When the USB device is plugged into the host, powered up by the host and the USB bus is reset by the host, the control endpoints become available for transfer of configuration details. Endpoints are also used to assign configuration descriptors and address to the device. (3)

Apart from this, the pipes have many characteristics associated to them. The pipes also have anyone type of data transfer associated to them. They have the directionality and bandwidth usage depending on the endpoints they are associated to. The pipe that connects the zero numbered endpoints of the USB device to the USB system on the host side is called the Default Control Pipe. Only after the endpoints zero are configured as mentioned above, the Default Control Pipe can be used for various configuration purposes by the client software in the host. After the USB device is configured using the Default Control Pipe, the other pipes become available. But still the USB System Software on the host retains the primary hold on the Default Control Pipe. (3)

Stream Pipe: Stream pipes deliver data in data packets with no USB-defined structures. The Stream pipe follows the First IN, First Out (FIFO) type of data transfer. The pipe is always connected to a particular endpoint of the USB device. The data that flows through the Stream pipe is always in the sequential manner and thus, there is no need to check for the order of the data at the host side. Only bulk, interrupt and isochronous transfer types are supported by the Stream pipes. (3)

Message Pipe: The interaction of the message pipes with the endpoints is quite different from the stream pipes. In this type of pipe, the host is the initiator of a transaction, i.e., the request is sent to device by the host and depending on the transaction type, the data is transferred in either of the direction. After some time, a status is provided. The request, data and status are all wrapped in a USB-defined structure and the message pipe uses this structure for communication. The perfect example of a message pipe is Default Control Pipe. Message pipe requests only a single endpoint number in both directions (IN and OUT) and doesn’t use two different direction endpoints with different endpoint numbers. Only control type transfer is supported by the message pipes. (3)

Pipes:

It is a connection between the software layer on the host side and the peer endpoints on the USB device side. A USB pipe represents the data flow between the two entities via temporary memory space in the host. The pipes interact in two different modes. They are as mentioned below:

- Stream pipe where the data being transferred does not use the USB-format structures.
- Message pipe where the data being transferred uses the USB-format structures.

The bus does not interpret the contents of data being transferred through the pipe. Even in the case of the message pipe, the bus fails to interpret the data being delivered although it in wrapped in the USB defined structures.
DATA TRANSFER TYPES IN USB

The USB is responsible for transferring data through a pipe between a memory buffer in the Software Client on the Host and the Endpoint on the Device side. The data transfer should be optimized to be compatible with the Client Software and the Endpoint. The data transfer types determine the characteristic of the communication flow. Some of these characteristics are as follows:

- Data Format Imposed by the USB
- Direction for Communication Flow
- Packet Size and Latency Constraints
- Required Data Handling
- Error Handling

USB supports 4 data transfer types:

- Control Transfers: This kind of data transfers is initiated by the Software to exchange control/status information. Mostly bursty and non-periodic in nature.
- Isochronous Transfers: Periodic and continuous data exchange to transfer time relevant information.
- Interrupt Transfers: Communication with bounded latency and low frequency.
- Bulk Transfer: Bandwidth independent, non-periodic data transfer with no bounds on time.

1. Control Transfer: Communication flows between the client software and the Control Endpoint use control transfer to exchange configuration/command/status type of information. Each USB device is has a Default Control Pipe (ENDPOINT 0) as a message pipe which is used by the USB system software to get access to the control, configuration and status details of the USB device. Normally a single control endpoint suffices the requirements for a device. Using the control transfer get access to the device descriptors and can modify device behavior. Control transfers are supported via bi-directional flow over message pipes (3). The control transfer takes place in either 2 or perhaps 3 stages:

   a. Set-up Stage: A specific request is made to the USB device (eg. To read the device descriptors) in the setup stage.
   b. Data Stage: This stage is required only for requests which require data transfer. For eg. A read descriptor request sends the descriptor contents during the data stage.
   c. Status Stage: This stage is required to deliver the result of the request.

2. Isochronous Transfer: Isochronous transfers in non – USB environment have the general implication of constant rate and error tolerant transfers. In USB environment, on a request for Isochronous transfer type provides the requester with:

   a. Guaranteed access to USB bandwidth with bounded latency.
   b. Guaranteed constant data rate through the pipe as long as data is provided to the pipe.
   c. No retrying of the attempts to deliver the data in case of delivery failure due to error.

An isochronous pipe is unidirectional. The endpoint description identifies whether the orientation of a pipe is either ‘IN’ or ‘OUT. If a device requires communicate in both directions then it has to establish two unique pipes for transfer either way.

3. Interrupt Transfer: These transfers are designed to support devices which send data infrequently but with bounded time constraints. A requester is provided with
the following on requesting a interrupt transfer pipe:
   a. Guaranteed maximum service period for the pipe.
   b. A provision for re-sending the data in case of failures.

Interrupt Transfers is implemented using POLLING. Each device is asked if it has any data to be transmitted. If it does not have any data to send it sends back a negative acknowledgement, indicating there is not data to be transferred. Data transfers are based on a scheduled basis. Interrupt transfers are based periodically such that, for a device overrun condition does not occur.

4. **Bulk Transfer**: The bulk transfer type is used by devices which need to transfer huge amounts of data with no time constraints and can operate on any bandwidth. Requesting this kind of pipe gives the requester:
   a. Access to USB on bandwidth available basis.
   b. In case of failures, a provision for re-sending the data.
   c. Guaranteed delivery but no guarantee of BW or latency.

Bulk transfer is BW dependent. On availability of large BW, the transfer completion is fast. When a lower BW is allocated for data transfer, the completion of data might take a longer time. A bulk pipe is a stream pipe and always has data transferred in either direction but not at the same time.

**USB SYSTEM**

The USB System can be shown as in the figure-5. The connection between a host and a device can be better explained as interactions with a numbers of layers and entities. This model basically gives a more of an overall logical view of the entire USB system. The layers can be explained as the following:

**The USB Bus Interface Layer:**

The actual connection between the host and the device is provided by this interface layer.

It provides signaling, physical and data/packet connectivity between them. The entity in the host-side layer offers both the hardware and the software that allows the USB device to get connected to the host.

**The USB Device Layer:**

This layer allows the host to perform the generic operations with the USB device. The connection between the entities at this layer is logical but the responsibility for the configurations of the USB device mainly lies in this layer.

**The Function Layer:**

This layer deals with the actual functionality of the device and its use on the host machine. The USB device can have one or more functionalities and for each of them there exist different client software that interacts with that functionality.

![Figure 5: USB Implementation (3)](image-url)
USB Host Interface

The USB host interface helps to facilitate USB communication between the client software and functionality on the USB device. Thus, it is necessary to understand the Host interface in detail. The figure-6 gives us better understanding of the host communication. There is only one host for each USB communication. The host has majorly the following entities:

- USB bus interface
- USB system
- Client

![Figure 6: USB Host Communication](image)

From the interconnect point of view, the USB bus interface is provided by both the USB device and the host through the Serial Interface Engine (SIE). But however, the bus interface on the host side has additional responsibilities as it has to perform the unique role of a host on the USB. This is implemented by the USB Host Controller. The Host Controller has an integrated Root Hub that provides attachment points to the USB devices via the USB wires. (3)

The USB System uses the Host Controller to transfer data between the host and the USB device. The interface between the USB System and the Host Controller depends on the hardware definition of the USB Host Controller and thus the USB System translates the data for the data transfer and the USB transactions as viewed by the client software. The USB System makes use of the wrapper USB structures for the communication with the Host Controller. It also manages the USB resources like the bus power and the bandwidth usage. The USB System communicates via the Default Control Pipe with endpoint zero of the USB device. This entity also configures the USB device to the host. This helps the client to access the USB whenever it wants to. The USB System has three main entities: Host Controller Driver (HCD), USB Driver (USBD) and the Host Software. HCD helps the mapping of the Host Controller Implementation into the USB System, so that the client can interact with the device without the knowledge of the Host Controller hardware to which the device is connected. The interface between the HCD and the USBD is never available directly to clients and this interface is defined by each operating system that supports various Host Controller implementations. (3)

The client layer describes the software that the entity uses to interact with the USB device and its functionality. When the device is attached to the host, the client should directly interact with the peripheral device. But due to the shared characteristics of the USB, the USB System Software is placed between the client and the device. Thus, the client cannot directly access the device hardware. The data that is transferred between the USB System and the Client Software is the data without any wrapper structures and the refined data from the USB device. The Client Software communicates with the USB device through different pipes and all these pipes together can be referred as an interface between the
Client Software and the functionality of the device. (3)

USB DRIVER ARCHITECTURE IN LINUX

We saw about the USB system on the host side. But it gave us a very generic view of how the softwares are interrelated. Now we need to look into the actual drivers and data structures used in USB to better understand the interface and the data flow. The figure 7 shows the architecture of the USB drivers in Linux operating system.

This architecture presents the Linux – USB subsystem and the constituent parts of this subsystem are (4):

- USB Core (usbc ore)
- Host Controller Driver (HCD)
- USB Client Driver
- USB file-system (usbfs)

**Host Controller Driver:**

This driver is the software that supports the host controller hardware and communicates with the USB core. (4) This driver helps to support the USB core such that the client driver can communicate with the USB device irrespective of the knowledge of the host controller type. This software is also connected to the hub driver which is the driver for the root hub. This hub driver helps monitor the devices attached to the root hub directly or indirectly. The detection and communication to all the devices on the bus which are not directly connected to the root hub but to intermediate hubs is the responsibility of the root hub and thus of the hub driver. Also the requests, data and the status that flow between the host and the device are all wrapped in USB-defined data structures (e.g.: urb). This helps different host controller drivers to support different controllers on the same operating system.

**USB Core:**

The USB core in the core layer in the USB subsystem and it is basically software that consists of codes of all the routines, macros and the data structures available to the HCDs and the client device drivers. (4) The USB core acts like a bridge between the client driver and the HCD. In order to be a bridge, the core provides a level of indirection that renders the client device drivers independent of the hardware-definition of the host controller used. This is done by providing two different APIs to both the HCD and the client driver. The API on the HCD side consist of all the routines and data structures which are all well-structured and contain all the information and necessary information about the USB device which are necessary for USB communication between the host and the device. The API on the client driver side doesn’t make use of the USB-defined macros and structures and thus, the data that flows between the core and client driver is in simple memory buffers.
**USB Client Driver:**

Each USB device has one or more functionalities and thus, to support and use those functionalities different softwares are needed. (4) The USB client driver is basically that software that corresponds to the functionality of the particular device. The operating system itself finds a matching client driver for the device. If the client software is not present in the system, it can be obtained from the USB device. Also, if the client driver is not provided, the host searches for a suitable driver for the device from other networks like local network, internet etc.

**USB File-system:**

If the user would like to access the USB device, the need for writing the kernel code is not required. The USB filesystem ubsfs is basically software in the kernel space that allows the user to access the data and the device from the user space. The file for a particular device in the filesystem is dynamically generated when the device is attached and configured to the host.

**Khubd:** This is basically a helper kernel thread that is used during the process of enumeration. The process of detecting port status changes and configuring the hot-pluggable devices is time-consuming. Thus the hub driver wakes up the khubd on detection of port status change.

**PLUG N PLAY**

Hot plugging is yet another important feature adherent to USB. This feature allows the host to automatically detect the attachment of a USB device to one of the ports and also install relevant software to access the device. It overcomes the problem of setting up switches and jumpers in order to configure a new device attached to one of its ports. Also it eliminates the need to restart a system on connection of a new device. Hence, in essence when a peripheral is connected it is immediately ready for use. Hence, this feature proves very user friendly.

**Enumeration of USB Device in Linux Systems:**

The attachment and detachment of a USB device to a port is recognized by the host using a process called as ENUMERATION. When a USB device is connected to a port, the following action is taken (3):

- The hub to which the USB device is connected reports the host of a new event by changing the status on the control pipe. The port still is unattached at this point and the device is in powered state.
- Then, the host requests for information on the exact nature of change to the hub and awakens a worker thread ‘khubd’.
- Now, the host identifies the port to which the device is connected via the khubd, the host then waits for 100 ms for the completion of the insertion process and the power at the device to become stable.
- The host then enables the port to which the device is connected and then resets the port.
- When the port is reset the port is also enabled at that instant.
- After reset, all its registers and states have been reset and it then answer to the default address.
- Before, the USB device get a permanent address, the default control pipe is still accessible via the default address.
- Here, the host collects all the device description details.
- A device can support a number of different configurations. The host then reads the configuration details of the device by checking each configuration. This process might take several milliseconds to complete.
• The USB device is allocated a configuration value depending on the configuration details of the transfer and it is also allocated a respective driver to handle the device. In this configuration state all the relevant endpoints have taken their characteristics.

• The 'khubd' then invokes the probe function of the driver and from this point the device is responsible for its normal operation.

• This completes the Enumeration process for the device.

IMPLEMENTATION

After learning about the entire USB protocol, we looked into the source code and found man data structures used in the Linux drivers for USB communication. To demonstrate the working of the USB data-structures and routines, we have implemented an example driver code (my_usb.c, attached along with the report) which aims at demonstrating the read and write data-transfer operations to and from the USB device. This code does not take into account the file system and memory management of the flash drive.

Brief Description of the Data Structures used in the Program: For the read and write operations, we have referred to a function usb_bulk_msg(). This function is used to transfer data between the user and USB device. Function usb_bulk_msg() takes important parameters like usb_device (which contains the details about the USB device), the pipe address, which is obtained from the endpoint number and other details like the data flow direction and the buffer (message to be sent to the device).

This is a wrapper function used to send messages in the USB-format. The data is wrapped into structure called the urb. The urb is the most important part of data transfer in USB. It basically forms an element of the USB stack. The urb consists of usb_device information, Pipe information, associated Data Buffer, DMA address for transfer, polling interval for interrupt or isochronous transfers, reference to the specific driver context, callback function handler and many other useful information for data transfer. The urb has to go through three important steps to complete the data flow: create, populate, and submit. All this steps are done in the usbcour and the client driver just calls the function usb_<transfertype>_msg().

In order to create a urb, usb_alloc_urb() is used allocate memory to the urb and also initializes the kernel object related to the urb and also the spinlock used to protect the urb. The populate process is done by the function call usb_fill_bulk_urb() in which the urb is filled with the data as mentioned above. The last stage, submission of the urb is done by usb_submit_urb() function. When the urb is submitted the data transfer takes place and this process is quite asynchronous and the callback handler is called after the submission of the urb to check the submission status and freeing the data buffers used. There are usb_free_urb() and usb_unlink_urb() used to delete the urb and to cancel any pending urb respectively. (4)

SELF-EVALUATION

The USB protocol is made ‘easy to use’, but the implementation is not that simple. This project gives an exhaustive explanation of the working of the USB protocol and communication. Thus, the project helped us to learn the basics of the USB interface and develop some programming skills for USB devices and drivers.

REFERENCES


[5] USB System Architecture (USB 2.0) – Don Anderson
[7] wiki.osdev.org/Universal_Serial_Bus