ENGINEERING AND R&D SERVICES

HCL

Develop USB Device Access Using LIBUSB, And Use Anywhere

WHITE PAPER

e)² Engineering experiences

Acceleration | Value | Technology

Table of Contents

Abstraction

We often come across situations where a USB device which runs perfectly on a Windows platform does not even get detected on Linux. A lack of support for USB devices on multiple platforms is one of the reasons. In order to support USB devices on multiple Operating Systems, vendors need to implement multiple OS dependent USB drivers.

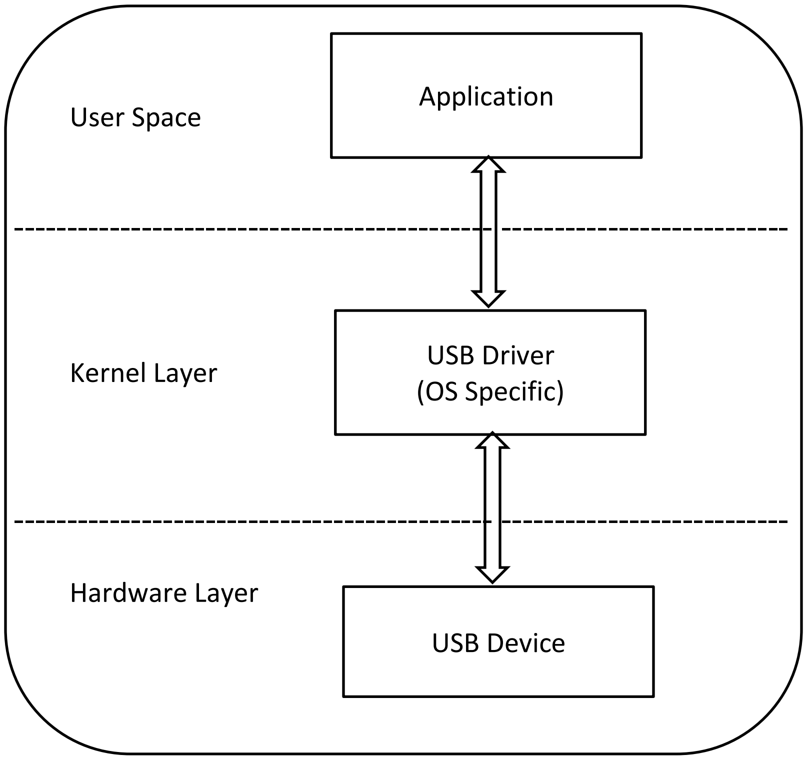
LIBUSB is a cross-platform user-mode library that provides access to USB devices. It eliminates the need to develop various USB drivers for various platforms for the same device, and makes life easier for device vendors.

The objective of this technical white paper is to provide insights on

1. USB basics
2. USB vendor challenges/expectations
3. LIBUSB
4. LIBUSB integrtation with Applications

The target audience of this paper would be developers who want to develop cross-platform compliant device drivers.

The figure below depicts the user space application communication with USB devices across platforms.

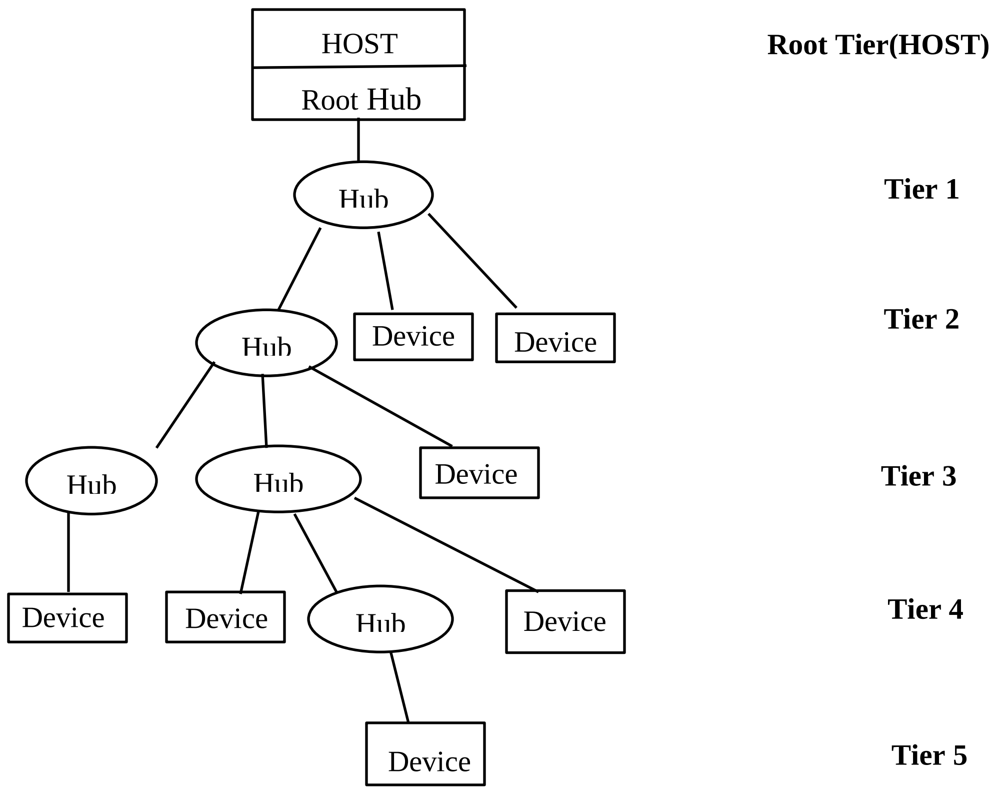


USB Basics

Introduction

The universal serial bus (USB) is a connection between a host computer and a number of peripheral devices using a single bus type.

The USB is actually an addressable bus system, with a seven-bit address code. It can support up to 127 different devices or nodes and there is a single host controller, which enumerates all the connected devices/hubs. A device can be plugged into a hub, and that hub can be plugged into another hub, and so on. However, the maximum number of tiers permitted is six.



USB HOST Responsibilities

All communications on this bus are initiated by the host. There can be no communication directly between USB devices that are connected to the same system. The host's responsibilites are:

1. Providing power, and detecting the attachment and removal of USB devices
2. Managing control/data flow between the host and USB devices
3. Collecting status and activity statistics

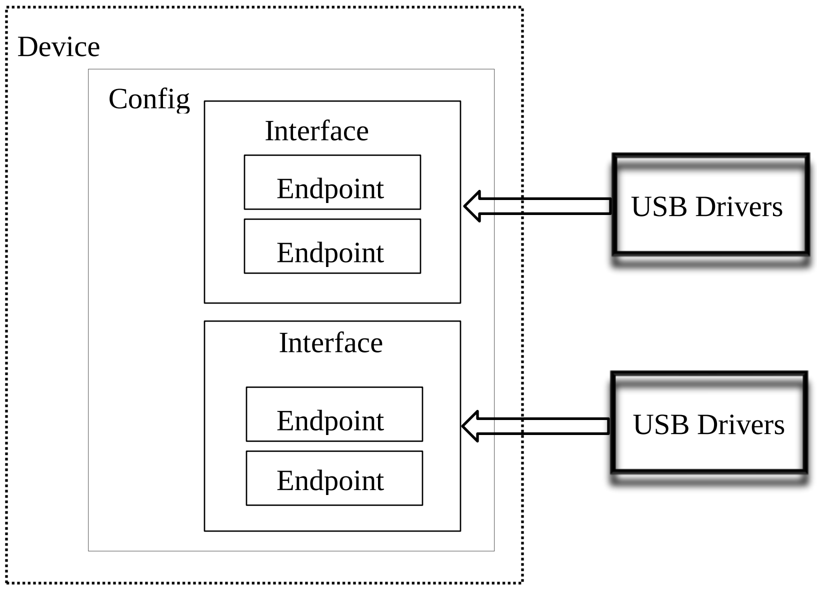
USB specifications

A USB device provides information about itself in data structures called USB descriptors. Descriptors hold information such as, the type of device, who makes it, what version of USB it supports, how many ways it can be configured, the number of endpoints and their types, etc.

Here are different sections of descriptors:

1. **Device Descriptors: A** USB device can only have one device descriptor. It specifies important information about the device, such as the supported USB version, maximum packet size, vendor and product IDs, and the number of possible configurations the device can have.
2. **Configuration Descriptors:** A USB device can have one or more configurations. The configuration descriptor specifies how the device is powered, what the maximum power consumption is, and the number of interfaces it has.
3. **Interface Descriptors:** The interface descriptor could be seen as a header or grouping of the endpoints into a functional group, performing a single feature of the device.
4. **Endpoint Descriptors:** A USB endpoint can be one of four different types that describe how the data is transmitted:
   a) **Control:** Control endpoints exchange configuration, setup, and command information between the device and the host.
   b) **Interrupt:** Interrupt endpoints are used by peripherals exchanging small amounts of data that need immediate attention.
   c) **Isochronous:** Isochronous endpoints are used by time critical streaming devices such as speakers and video cameras.
   d) **Bulk:** Bulk endpoints are used by devices like printers and scanners that receive data in one big packet. Here, timely delivery is not critical.

The figure below depicts binding between descriptors and drivers.



USB Device Driver Development Phases

Writing of USB device drivers requires special skills and is quite a lengthy process, which must be repeated for every new operating system to be supported. The basic driver development process is as follows:

1. Learn the internals of the operating system to be supported (Windows/Linux/VxWorks…)
2. Learn how to write a device driver on this operating system (for example DDK)
3. Write the kernel mode device driver which does the basic hardware input/output (in kernel mode)
4. Write the application in user mode, which accesses the hardware through the device driver written in kernel mode
5. Repeat steps 1-4 for each new operating system on which the code should run.

USB Device Vendor Challenges and Expectations

Vendors face the following challenges in supporting USB devices across platforms:

1. To develop multiple USB Device Drivers for various Operating Systems – a complex and time consuming task
2. Require a lot of effort to understand kernel level modules, system internals, etc for each operating system
3. To maintain multiple code base, release updates, bug tracking, etc. – which increases the cost
4. The need for good and effective tools for
   a. Debugging the code
   b. Memory optimizations, and code coverage

Vendors always expect the following, while developing new devices:

1. Saving cost/time involved in different phases of product development and quickly bringing the product to market
2. Avoiding complexity in implementing the drivers and maintenance of the code base
3. Vendor specific devices
4. Re-using the code base for new products wherever applicable

One Solution for All Expectations

LIBUSB has been designed to address all the above challenges. Here are its major advantages:

1. LIBUSB is a high-level language API which conceals low-level kernel interactions with USB modules
2. LIBUSB library functions provide high level abstraction to kernel structures
3. Simplified interface allows developers to develop USB drivers from the userspace

About LIBUSB

LIBUSB is a C library that allows you to communicate with USB devices from userspace. It is intended for use by developers, to facilitate the production of applications that communicate with the USB hardware.

1. **It is portable**: Using a single cross-platform API, it provides access to USB devices on Linux, OS X, Windows, Android, OpenBSD, etc.
2. **It is user-mode**: No special privilege or elevation is required for the application to communicate with a device.
3. **It is version-agnostic:** All versions of the USB protocol, from 1.0 to 3.0 (latest), are supported

LIBUSB Features

The beauty of LIBUSB lies in its cross-platform functionality. The driver written for one platform can be easily ported onto another platform, with little or no changes.

The following are the major features of LIBUSB:

- All transfer types are supported (control/bulk/interrupt/isochronous)
- Two transfer interfaces: Synchronous (simple) and Asynchronous (more complicated, but more powerful)
- Thread safe
- Compatible with libusb-0.1 through the libusb-compat-0.1 translation layer
- Hotplug support

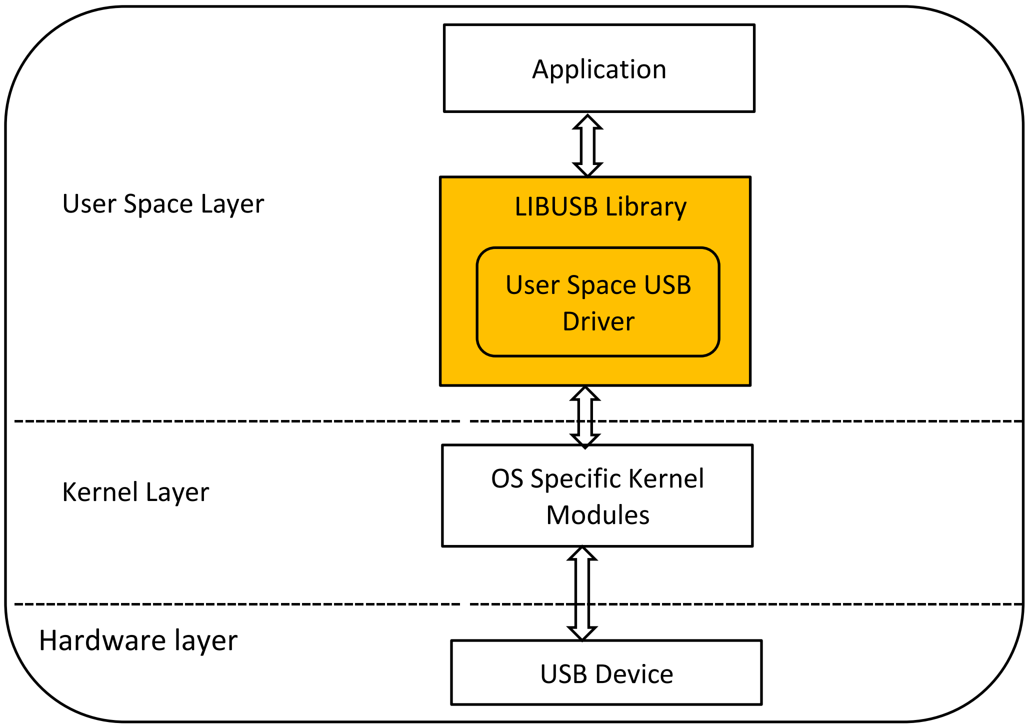


Figure: Userspace application communication with USB devices in cross platforms using LIBUSB

Development Environments

- Linux
    - Any modern Linux system with usbfs.1
- Mac OS X
    - Any modern version of Mac OS X (PowerPC or x86 either in 32 and 64 bits)
- Windows
    - MinGW (32 bit) or MinGW-w64. To compile a library that is both 32 and 64 bit compatible, you should use a version of MinGW-w64 that supports both -m32and -m64 (a.k.a. "multilib"). A pre-built multilib version can be downloaded withTDM64
    - Microsoft Visual C++ (Visual Studio)
    - Windows DDK build environment
    - Device driver support: WinUSB, HID, libusb-win32 (libusb0.sys) and libusbK
- OpenBSD and NetBSD
    - Only device with ugen driver are supported

Language Binding

- Java Applications
- Python Applications
- C#, .Net Applications
- C, C++ Applications
- Ruby Applications

LIBUSB Integration with Applications

LIBUSB provides APIs to access USB devices from userspace. These APIs can be used by application developers to implement the required functionalities. Here are a few use cases for how to integrate the LIBUSB APIs in userspace applications.

Use Case 1: LIBUSB Initialization/ Deinitialization:

1. Initialization must be performed before using any LIBUSB functionality
2. One must not call any libusb functions after deinitialization of LIBUSB
3. By default, logging is set to NONE and can be enabled using libusb_set_debug() function for certain messages. Can be called any time after initialization

Algorithm:

```
int  result;
libusb_context *ctx = NULL;
Step-1:  result = libusb_init(ctx);
Step-2:  if (result  <  0)
             return result;
            .
            .
Step-m: libusb_set_debug(NULL, LIBUSB_LOG_LEVEL_INFO);
            .
           Do the Required Operations
            .
Step-n:  libusb_exit(ctx);
```

Use Case 2 : USB Device Enumeration, Open and Close required Device

This use case is designed to help with the following operations:

1. Enumerating the USB devices currently attached to the system
2. Choosing a device to operate from your software
3. Opening and closing the chosen device

Algorithm:

```
 libusb_device **list;
 libusb_device *found = NULL;
 ssize_t i = 0;
 int err = 0;
libusb_context *ctx;
Step-1: Initialize libusb
Step-2:  ssize_t cnt = libusb_get_device_list(ctx,  &list);
Step-3:  if (cnt < 0)
             error();
Step-4:    for (i = 0; i < cnt; i++) {
               libusb_device *device = list[i];
               if (is_interesting(device)) {
                  found = device;
                  break;
               }
           }
Step-5:if (found) {
           libusb_device_handle *handle;
```

```
                err = libusb_open(found, &handle);
                if (err)
                        error();
        }
        Step-6: libusb_free_device_list(list, 1);
```

Key Points:

1. Discover devices using libusb_get_device_list().
2. Choose the device that you want to operate, and call libusb_open().
3. Unref all devices in the discovered device list.
4. Free the discovered device list.
5. The order is very important - do not unreference the device before attempting to open it, because unreferencing it may destroy the device.

Use Case 3: USB data transfers:
libusb offers two separate interfaces for device I/O.

1. Synchronous I/O
2. Asynchronous I/O

At a logical level, the USB transfers typically happen in two parts. For example, when reading data from an endpoint:

- A request for data is sent to the device
- Some time later, the incoming data is received by the host

Or when writing data to an endpoint:

- The data is sent to the device
- Some time later, the host receives acknowledgement from the device that the data has been transferred.

The main difference is that the synchronous interface combines both steps indicated above into a single function call, whereas the asynchronous interface separates them.

3.1 Synchronous I/O
The synchronous I/O interface allows you to perform a USB transfer with a single function call. When the function call returns, the transfer is completed and you can then parse the results.
Algorithm:

```
        unsigned char data[4];
        int actual_length;
        Step1: int r = libusb_bulk_transfer(handle, LIBUSB_ENDPOINT_IN, data, sizeof(data), &actual_length, 0);
        Step2: if (r == 0 && actual_length == sizeof(data)) {
                // results of the transaction can now be found in the data buffer
                // parse them here and report
                } else {
                        error();
                }
```

Key Points:

1. The main advantage of this model is simplicity: you can do everything with a single simple function call.
2. Use synchronous I/O only for simple transfers

Synchronous I/O Limitations:

1. The application will sleep inside libusb_bulk_transfer() until the transaction is completed. If transfer takes 3 hours, the application will sleep for that long. Execution will be tied up inside the library and the entire thread will be useless for that duration.
2. By tying up the thread with that single transaction, there is no possibility of performing I/O with multiple endpoints and/or multiple devices simultaneously, unless you resort to creating one thread per transaction.
3. There is no opportunity to cancel the transfer after the request has been submitted.

3.2 Asynchronous I/O

The asynchronous interface is built around the idea of separating transfer submissions and handling of transfer completion (on the other hand, the synchronous model combines both of these into one). There may be a long delay between submission and completion. However, the asynchronous submission function is non-blocking, so it will return control to your application during that potentially long delay. It solves all the limitations in Synchronous I/O.

1. LIBUSB's asynchronous interface presents the non-blocking functions which begin a transfer and then return immediately, instead of functions moving to blocked state until the I/O is completed.
2. Application passes a callback function pointer to this non-blocking function, which LIBUSB will call with the results of the transaction when it has been completed.
3. Transfers which have been submitted through the non-blocking functions can be cancelled with a separate function call.
4. The non-blocking nature of this interface allows you to simultaneously perform I/O to multiple endpoints on multiple devices, without having to use threads.

For asynchronous I/O, LIBUSB implements the concept of a generic transfer entity for all types of I/O (control, bulk, interrupt, isochronous). The generic transfer object must be treated slightly differently depending on which type of I/O you are performing with it.

1. Allocate a libusb_transfer
2. Populate the libusb_transfer instance with information about the transfer you wish to perform
3. Ask libusb to submit the transfer
4. Examine transfer results in the libusb_transfer structure
5. Clean up resources

Algorithm:

Step-1: Allocate, populate and submit the libusb_transfer

```
void myfunc() {
```

```
                struct libusb_transfer *transfer;
                unsigned char buffer[LIBUSB_CONTROL_SETUP_SIZE] __attribute__ ((aligned (2)));
                int completed = 0;
                transfer = libusb_alloc_transfer(0);
                libusb_fill_control_setup(buffer, LIBUSB_REQUEST_TYPE_VENDOR |
                                          LIBUSB_ENDPOINT_OUT, 0x04, 0x01, 0, 0);
                libusb_fill_control_transfer(transfer, dev, buffer, cb, &completed, 1000);
                libusb_submit_transfer(transfer);

                while (!completed) {
                        poll(libusb file descriptors, 120*1000);
                        if (poll indicates activity)
                                libusb_handle_events_timeout(ctx, &zero_tv);
                }
                printf("completed!");
                // other code here
        }
```

Step-2: Call Back function registered with libusb_transfer structure

```
        void cb(struct libusb_transfer *transfer)
        {
                int *completed = transfer->user_data;
                *completed = 1;
        }
```

Step-3: Handle Events thread for tracking submitted transfer status and invoking the registered callback

```
        void *event_thread_func(void *ctx)
        {
          while (event_thread_run)
            libusb_handle_events(ctx);

          return NULL;
        }

        void  libusb_handle_events(void *ctx)
        {
                //some code
                //track the transfer status
                //Invoke the callback with transfer status – success/fail/cancled
                // set event_thread_run to 0
        }
```

Conclusion

Supporting a USB device across platforms is a difficult task, and expensive for device vendors, and especially for device enumeration/data transfer related implementations. Maintenance of the code base is also a big challenge for vendors.

Vendors can avoid all these complex implementation tasks and yet support multiple platforms with LIBUSB, as it takes care of all kernel level interactions. As a result, it offers the following advantages:

1. Saves significant implementation time
2. Its lean code base makes maintanence easy
3. Gets the product to market sooner at minimal cost

References

- www.usb.org
- www.libusb.sourceforge.net/

A Note of Thanks

Special thanks to Sridhar Chebrolu (sridhar_veda@hcl.com) for his guidance..

Author Info

Nagendra Simhadri

Nagendra is a Senior Technical Lead in HCL's Engineering and R&D Services Group. He has more than 10 years of industry experience in Mobile Engineering, Servers and Storage, Network Security Domains, and Technical Pre-sales in Mobile Technologies. He is experienced in C/C++, Data Structures, Algorithms, Templates and STL, PCI and PCIe on Linux and Windows platforms.