Developing a driver for a film scanner by means of USB sniffing and reverse engineering

Hugo Platzer

University of Salzburg

December 14, 2017

1 The USB standard

1.1 Motivation

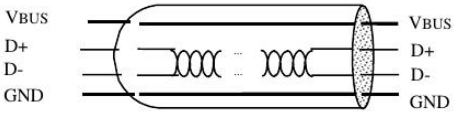
USB is an interface intended to connect various peripherals to PCs. These include: Human interface devices like keyboards and mice; storage devices like card readers, external hard disks, memory sticks and smartphones; multimedia devices like microphones, speakers, cameras and scanners. Some highlights leading to its wide adoption [3, p. 11]:

- Unified interface for all kinds of peripherals
- Plug and play: The user plugs in the device, the configuration (e.g. loading the appropriate drivers) is done automatically by the operating system
- Number of ports can be increased using hubs. Multiple hubs can be chained allowing for up to 127 devices on a single root port.
- High data rate of 480 Mbit / s (USB 2.0 high-speed mode). Also offers low-latency transfers for real-time audio/video applications
- Backwards compatibility: Older USB 1.1 devices can be used at 2.0 hosts. High-speed USB 2.0 devices can also be used on older machines supporting only USB 1.1 (albeit at lower speed).

1.2 Electrical side

A USB cable has four wires: One as ground, VBUS for a 5 V power supply and two for data transmission. The power line allows it to draw up to 100 mA without any configuration. This is useful for simple devices that are not using the data lanes, just the power, like USB lights. Also it allows for devices not taking much power to be self-powered which eliminates the need for an extra

Figure 1: USB cable cross-section [3, p. 17]



power supply and connector. Devices can ask the host for more power (up to 500 mA), those that need even more (like a lot of scanners) need an external supply. [3, p. 17f.]

1.3 Signaling

1.3.1 Low-level states

USB is a serial bus which means there is only a single path for data transmission. Differential signalling across the D- and D+ wires is used, which means the difference in voltage across the two wires (rather than some absolute) determines the state. This is beneficial because noise during transmission should affect both lines equally, not changing the difference. Higher frequencies and thus data rates become possible.

Table 1: USB speed modes [3, p. 159]

Mode	Speed	Bit time
Low Speed	1.5 Mbit / s	667 ns
Full Speed	12 Mbit / s	83 ns
High Speed	480 Mbit / s	2 ns

Table 2: Low-level data line states (only applies to Full Speed) [3, p. 145]

Levels	State
Differential '0'	D- high, D+ low
Differential '1'	D- low, D+ high
Single Ended Zero (SE0)	both low
Single Ended One (SE1)	both high (illegal state, should never happen)
Data 'J' state	Differential '1'
Data 'K' state	Differential '0'
Idle state	Data 'J' state
Start of Packet (SOP)	Switch from idle to 'K'
End of Packet (EOP)	SE0 for 2 bit times followed by 'J' for 1 bit time
Disconnect	SE0 for $\geq$ 2 us
Connect	Idle for 2.5 us
Reset	SE0 for $\geq$ 2.5 us

Low Speed is used for devices where speed is not important (mice, keyboards). It allows for cheaper cables and electronics. High Speed is only available in USB 2.0. For reasons of simplicity, only full speed signaling will be covered here. [3, p. 12]

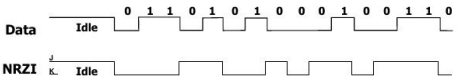
SE0 is the state of the data lines if no device is connected. The host recognizes a device being plugged in by the D+ line being "pulled up" to high. It then most likely initiates a reset so the device is in a known state for communication to begin. Similarly, a disconnect is sensed by a SE0 for some time. [3, p. 149]

It is important to note that **all communication on the USB bus is initiated by the host**. Devices on the bus can not directly talk to each other and can only talk to the host as a direct response to a request made by it before. [3, p. 27]

1.3.2 Bitstream encoding

USB uses NRZI encoding for the transmitted data: A zero is represented by a change to the opposite state while a one is represented by staying in the same state.

Figure 2: NRZI bitstream encoding [3, p. 157]



For keeping the receiver clock in sync with the data it is not ideal if the signal stays at J or K for too long. To prevent this, a technique called "bit stuffing" is used: Before doing NRZI encoding, a zero is inserted after every six consecutive ones in the data. The receiver recognizes the stuffed bits during decoding and discards them. [3, p. 157]

1.4 Packets

Packets are the atomic unit of data transmission in USB. In between packet transmission, the bus remains in an idle state. Every packet starts with a sync pattern to synchronize the clocks between sender and receiver. Next are the actual data bits. The packet is terminated by an EOP state. Fields in a packet are transmitted least-significant bit first. [3, p. 195]

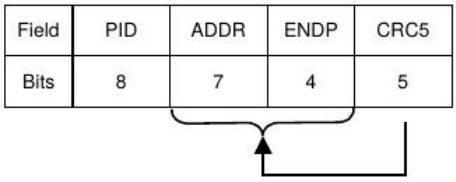
The first 8 bits of every packet contain the packet ID (PID) which identifies its type and thus how the rest of the packet data should be interpreted. The PID is 4 bits long, they are transmitted a second time in reverse bit order to allow the receiver to quickly discard a faulty packet. There are 17 different packet types (PRE and ERR have the same ID, some are only relevant for High Speed) [3, p. 195]:

Table 3: USB packet types; notice how the least-significant two bits identify the packet category [3, p. 196]

PID type	PID name	PID bits (3..0)	Description
Token	OUT	0001	Address + endpoint number for host-to-device transaction
	IN	1001	Address + endpoint number for device-to-host transaction
	SOF	0101	Start-of-frame marker, frame number
	SETUP	1101	Special host-to-device transaction for device configuration
Data	DATA0	0011	Data packet
	DATA1	1011	Data packet
	DATA2	0111	Data packet (only High Speed)
	MDATA	1111	Data packet (only High Speed)
Handshake	ACK	0010	Receiver accepts error-free data packet
	NAK	1010	Receiver cannot accept data or transmitter cannot send data
	STALL	1110	Endpoint halted or control pipe request not supported
	NYET	0110	Data packet (only High Speed)
Special	PRE	1100	Preamble to enable downstream traffic to low-speed devices
	ERR	1100	Split Transaction error handshake
	SPLIT	1000	High speed Split Transaction token (only High Speed)
	PING	0100	High speed control flow probe (only High Speed)
	Reserved	0000	Reserved PID

1.4.1 Token packet

Figure 3: Token packet format [3, p. 199]

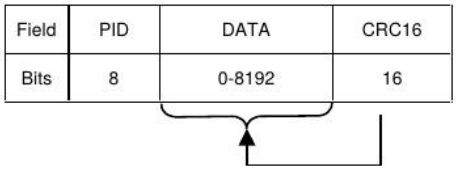


Token packets are used at the start of so-called transactions to specify the target of the transaction on the bus, namely a certain device and endpoint. There are 127 possible devices on a bus (address 0 is reserved for a device that has not been configured yet). [3, p. 256]

Endpoints are logical entities on a device that are used as sources and sinks of data in so-called pipes. A pipe is either in OUT (to device) or IN (to host) direction. Endpoint 0 is a special bidirectional pipe that must be available on every device right after the reset. It is used mainly for identifying and configuring the device. [3, p. 33]

1.4.2 Data packet

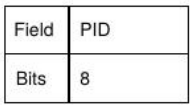
Figure 4: Data packet format [3, p. 206]



Used to transmit the actual data in a transaction.

1.4.3 Handshake packet

Figure 5: Handshake packet format [3, p. 206]



Used to report the status of a transaction.

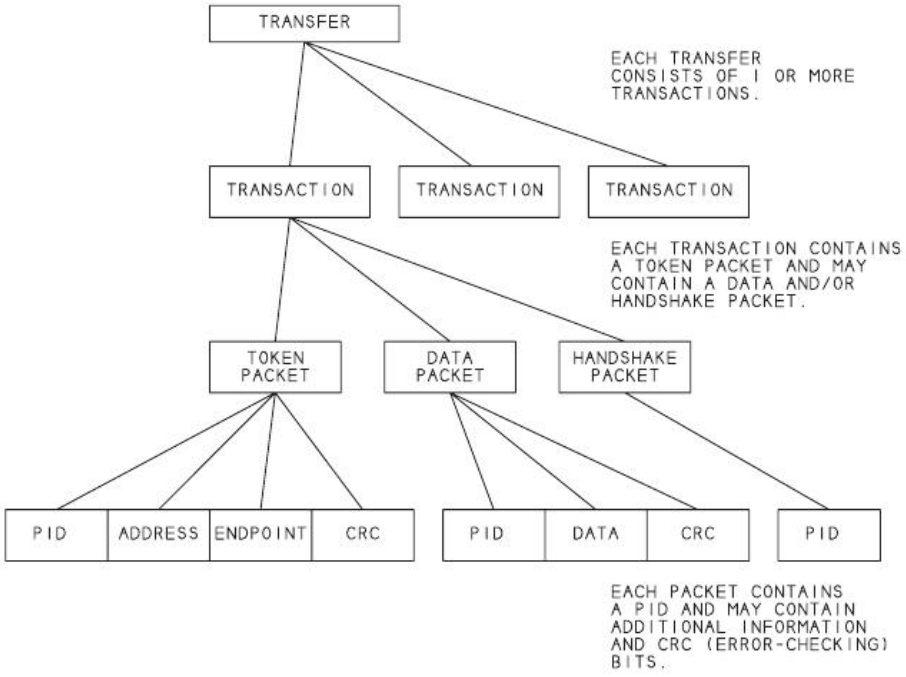
1.5 Transfers

Transfers are the abstraction level of data exchange as seen at the software side of the host. From the host's perspective, a transfer transmits a string of bytes from / to the device. A transfer is directed to one of the device's endpoints, each of them has one of four transfer types plus a transfer direction asscociated with it during device configuration. Transfers are composed of transactions, which usually consist of three packets [3, p. 209ff.]:

1. A token packet to tell the type of the transaction and its destination (as all USB communication is initiated by the host, the token packet always comes from the host)
2. A DATA packet for the actual payload. Length can vary between 8-64 bytes on Full Speed links. This can go host to device or device to host.
3. A handshake packet (usually ACK, NAK) lets the sender know if the data was received successfully.

Transfers usually consist of multiple transactions.

Figure 6: Composition of a USB transfer [1, p. 44]



1.5.1 Control transfer

Every device must have endpoint 0 for control transfers. Control endpoints are message-based. This means each transfer is for a specific message which has defined length, format and purpose. There can be transfers in either direction on the same endpoint, although each transfer has a specified (data stage) direction. Control transfers start with a Setup stage / transaction specifying the target device and endpoint plus an extra 8 bytes of data called the Device Request. [3, p. 38f.]

Table 4: Device Request format [3, p. 248]

Offset (bytes)	Field	Size (bytes)	Description
0	bmRequestType	1	Bits 0..4: Recipient 0 Device 1 Interface 2 Endpoint 3 Other 4..31 Reserved Bits 5..6: Type 0 Standard 1 Class 2 Vendor 3 Reserved Bit 7: Transfer direction 0 Host to device 1 Device to host
1	bRequest	1	Specific request
2	wValue	2	Varies according to request
4	wIndex	1	Varies according to request, typically used for a index or offset
6	wLength	2	Number of bytes to be transferred in the data stage (0 means no data stage)

Depending on the kind of control transfer, there may be a data stage consisting of data transactions, but this is not required. If there are data transactions, all are going in the same direction as specified by the request type. The transfer is completed with a Status transaction to report back about the success of the whole transfer.

Control transfers are used right after the device reset to get information about its capabilities and set a configuration. For this, there are 11 bRequest values called Standard Requests that have to be supported by all devices as part of the standard.

Table 5: Some important Standard Requests [3, p. 250]

bRequest	bmRequestType (7..0)	wValue	wIndex
GET_DESCRIPTOR	10000000	Descriptor type and index	Zero or Language ID
SET_ADDRESS	00000000	Device address	Zero
GET_STATUS	10000000 10000001 10000010	Zero	Zero Interface Endpoint
SET_CONFIGURATION	00000000	Configuration value	Zero

bRequest	wLength	Data
GET_DESCRIPTOR	Descriptor length	Descriptor
SET_ADDRESS	None	None
GET_STATUS	Two	Device, Interface or endpoint status
SET_CONFIGURATION	None	None

They are also used to control the device after the configuration phase during normal operation. There are standardized requests (Class Requests) for certain device classes like keyboards, storage devices etc.

Other non-standard devices have custom, vendor-specific requests that are not documented in the USB standard.

1.5.2 Bulk transfer

Bulk transfers are used to transmit large amouts of data with guaranteed delivery and low overhead but no latency / bandwidth constraints. Compared to control transfers, they are stream-based. This means there is no defined message format or size. For an IN endpoint, the host would poll the device endpoint for data until it receives (cumulative) as many bytes as requested by the software. A transfer can also be ended by the device sending a data transaction of less size than the maximum of this endpoint (data is always split so there is at most one smaller packet at the end) or a zero-size data transaction. To avoid data packets being lost, they alternate between the DATA0 / DATA1 PIDs. A single bulk endpoint is only for incoming or only for outgoing transfers. The host schedules them at times when the bus bandwidth is not used by other transfer types. Typical uses for bulk transfers are transferring data from/to an external harddrive, to a printer or from a scanner. [3, p. 52ff.]

1.5.3 Interrupt transfer

On the bus, interrupt transfers look the same as bulk transfers. The main difference is scheduling: The host guarantees that a transfer attempt is made as often as specified in the endpoint descriptor (1 - 100 ms). Despite the name, interrupt transfers have nothing to do with hardware interrupts. All communication on the USB bus is initiated by the host, so it has to poll the device for data. Typical uses for interrupt transfers are receiving keypresses from keyboards and movements from mice. [3, p. 48ff.]

1.5.4 Isochronous transfer

Isochronous transfers are for transmitting data at a constant rate with guaranteed latency. Compared to interrupt transfers, they offer a guaranteed transfer rate (with interrupt transfers the interval between two transfers can be anywhere between zero and the specified maximum). The drawback is a lack of any handshake / retry mechanism. Typical uses for isochronous transfers are USB sound cards or webcams. [3, p. 44ff.]

1.6 Device initialization

The process from a device being plugged in to it becoming usable for the user roughly goes as follows (every USB bus has at least a root hub device interacting with the operating system) [1, p. 87ff.]:

1. The hub detects the device by the change in levels on the data lines.
2. The hub reports the device to the host.
3. The host tells the hub to reset the device so communication can begin.
4. The host assigns an address to the new device.
5. The host asks for the Device Descriptor to get information identifying the device (vendor / product ID, device class).
6. The host looks for a driver handling this VID / PID combination, if it does not find one, it leaves the device in an unconfigured state.
7. If a driver is found, it is loaded. The device is asked for its configuration profiles. The driver sets a configuration and can now make the device serve its purpose.

1.6.1 Device descriptor

After setting the address, a GET_DESCRIPTOR device request is made, asking for the device descriptor.

Table 6: Device descriptor format [3, p. 262f.]

Offset (bytes)	Field	Size (bytes)	Description
0	bLength	1	Descriptor size in bytes
1	bDescriptorType	1	1 for DEVICE descriptor
2	bcdUSB	2	USB specification release number
4	bDeviceClass	1	Class code
5	bDeviceSubclass	1	Subclass code
6	bDeviceProtocol	1	Protocol code
7	bMaxPacketSize0	1	Maximum packed size for Endpoint 0
8	idVendor	2	Vendor ID
10	idProduct	2	Product ID
12	bcdDevice	2	Device release number
14	iManufacturer	1	Index of manufacturer string descriptor
15	iProduct	1	Index of product string descriptor
16	iSerialNumber	1	Index of serial number string descriptor
17	bNumConfigurations	1	Number of possible configurations

bDeviceClass Classes are types of devices that were standardized in USB. There are class codes for audio devices, human interface devices, storage devices etc. There is also a vendor-specific class code for devices not conforming to a standardized class. They are further divided into subclasses specifying the actual functionality more precisely (i.e. keyboard vs. mouse for HID device).

idVendor is a 16-bit identifier assigned by the USB Implementers Forum for each manufacturer of USB compliant devices. Manufacturers must apply for a unique ID there.

idProduct is a 16-bit identifier assigned by the manufacturer of the device. Manufacturers manage their own PID space so that IDs are unique per device.

1.6.2 Configuration descriptor

Configurations are profiles that define one or more interfaces for the device and their characteristics. Only one configuration can be active at a time. [3, p. 244]

Table 7: Configuration descriptor format [3, p. 265]

Offset (bytes)	Field	Size (bytes)	Description
0	bLength	1	Descriptor size in bytes
1	bDescriptorType	1	2 for CONFIGURATION descriptor
2	wTotalLength	2	Total length of all descriptors for this configuration
4	bNumInterfaces	1	Number of interfaces in this configuration
5	bConfigurationValue	1	Value to be used to select this configuration with SET_CONFIGURATION
6	iConfiguration	1	Index of string descriptor for this configuration
7	bmAttributes	1	Bitmap of configuration characteristics (Self-powered etc.)
8	bMaxPower	1	Maximum power consumption from bus in this configuration (mA)

1.6.3 Interface descriptor

Interfaces are sets of related endpoints that together provide a certain feature of the device. Deiices can have multiple active interfaces at the same time. Additionally, interfaces can have alternate settings that change its endpoints' characteristics. [3, p. 244]

Device drivers are often assigned to serve a specific interface rather than an entire device.

Table 8: Interface descriptor format [3, p. 268f.]

Offset (bytes)	Field	Size (bytes)	Description
0	bLength	1	Descriptor size in bytes
1	bDescriptorType	1	4 for INTERFACE descriptor
2	bInterfaceNumber	1	Number identifying this interface
3	bAlternateSetting	1	Value specifying alternate setting for this interface
4	bNumEndpoints	1	Number of endpoints in this interface (excluding default endpoint 0)
5	bInterfaceClass	1	Type of functionality provided by interface (similar to Device class)
6	bInterfaceSubClass	1	Subclass of device type (similar to Device class)
7	bInterfaceProtocol	1	Protocol of class-specific requests made for this interface
8	iInterface	1	Index of string descriptor for this interface

1.6.4 Endpoint descriptor

Endpoint descriptors specify an endpoint's address, type and polling interval.

Table 9: String descriptor format [3, p. 269ff.]

Offset (bytes)	Field	Size (bytes)	Description
0	bLength	1	Descriptor size in bytes
1	bDescriptorType	1	5 for ENDPOINT descriptor
2	bEndpointAddress	1	Bits 3..0 give endpoint number, bit 7 gives endpoint direction (0 - out, 1 - in)
3	bmAttributes	1	Bits 1..0 describe endpoint type (Control, Isochronous, Bulk, Interrupt). Bits 5..2 specify extra options for isochronous endpoints
4	wMaxPacketSize	2	maximum packet size of the endpoint
6	bInterval	1	How often the host should poll for data (in microframes)

1.6.5 String descriptor

String descriptors provide information about the device in human-readable form. String descriptor 0 transmits language codes supported by the device.

Table 10: String descriptor format [3, p. 273f.]

Offset (bytes)	Field	Size (bytes)	Description
0	bLength	1	Descriptor size in bytes
1	bDescriptorType	1	for STRING descriptor
2	bString	rest	String, UNICODE encoded

2 USB sniffing

USB sniffing is the process of recording communication on the USB bus. This is useful for device developers, driver developers, keyloggers etc.

2.1 Hardware sniffing

Hardware sniffing uses a dedicated device that sits on the bus in between the host and device. Dedicated software is used to interpret the low-level signals as packets / transactions.

Advantages of hardware USB analyzers [4]:

- Independent of the host, not affected by bugs in its USB hardware / software
- Allow analysis down to the lowest level (voltage levels between wires)
- See all packets transmitted, including unsuccessful transactions etc.

2.2 Software sniffing

Software sniffing does not use dedicated hardware to listen to the signals on the wires. Instead, the host operating system reports packets sent / received to some software running on the same host.

Advantages of purely software USB analyzers:

- Lower cost (free when using usbmon / Wireshark)
- Easy setup, convenient, no extra hardware required

2.2.1 usbmon

usbmon is a module for the Linux kernel that allows access to USB request blocks (URBs) as they are being processed. URBs loosely correspond to USB transfers. [2]

2.2.2 Wireshark

Wireshark is a cross-platform sniffing and packet analysis software that became famous as an Ethernet (network) traffic sniffer. Its configurable interface allows the dissection of raw captured packet bytes into more meaningful fields. Captures can be saved to disk for later analysis.

For sniffing USB traffic, it requires the *usbmon* kernel module plus suitable permissions (running it as root is simplest). On the main screen, one selects one of the host's USB buses for recording. [5] It is good practice to check whether any devices other than the one to be recorded are on the same bus. This can be done with the *lsusb* command, which lists all connected USB devices and what bus they are on.

2.3 Example

2.3.1 Scenario

To provide some insight into the sniffing process and how the recorded data can be interpreted, I chose to record keystrokes from a USB keyboard. Since USB keyboards are very widespread and follow a standard protocol, the experiment can be repeated by almost anyone.

A bus with no devices on it (except the mandatory root hub with address 1) is chosen for sniffing. Recording is started before the device gets attached. The keyboard is attached, then the following keys are pressed and then released (one after another): a, b, a. Recording is then stopped.

For brevity, only a few of the captured packets are discussed here.

2.3.2 Device configuration

When the device is connected, a process like in subsection 1.6 starts. The device is reset, an address issued and descriptors queried. The device descriptor and the interface / endpoint descriptor are shown here.

Figure 7: Device descriptor of USB keyboard

```
▾ DEVICE DESCRIPTOR
    bLength: 18
    bDescriptorType: 0x01 (DEVICE)
    bcdUSB: 0x0110
    bDeviceClass: Device (0x00)
    bDeviceSubClass: 0
    bDeviceProtocol: 0 (Use class code info from Interface Descriptors)
    bMaxPacketSize0: 8
    idVendor: China Resource Semico Co., Ltd (0x1a2c)
    idProduct: Unknown (0x0e24)
    bcdDevice: 0x0110
    iManufacturer: 1
    iProduct: 2
    iSerialNumber: 0
    bNumConfigurations: 1
```

The class code of 00 signifies that information about the device should be obtained from the interface descriptor. The vendor name is queried from a database by Wireshark (only the 2 byte ID is transmitted over the bus).

Figure 8: Interface / endpoint descriptor of USB keyboard

```
▾ INTERFACE DESCRIPTOR (0.0): class HID
    bLength: 9
    bDescriptorType: 0x04 (INTERFACE)
    bInterfaceNumber: 0
    bAlternateSetting: 0
    bNumEndpoints: 1
    bInterfaceClass: HID (0x03)
    bInterfaceSubClass: Boot Interface (0x01)
    bInterfaceProtocol: Keyboard (0x01)
    iInterface: 0
▸ HID DESCRIPTOR
▾ ENDPOINT DESCRIPTOR
    bLength: 7
    bDescriptorType: 0x05 (ENDPOINT)
  ▸ bEndpointAddress: 0x81  IN  Endpoint:1
  ▾ bmAttributes: 0x03
      .... ..11 = Transfertype: Interrupt-Transfer (0x3)
  ▸ wMaxPacketSize: 8
    bInterval: 10
```

The interface identifies as a keyboard and has one extra input endpoint. Interrupt transfers should be performed there every 10ms. The data received from the endpoint is an 8-byte HID report giving information about pressed keys. The first of these bytes describes pressed modifier keys (Ctrl, Alt etc.) the second byte is reserved, the third byte describes the actual pressed key. [6]

2.3.3 Data transmitted while keys are pressed

Figure 9: HID reports from USB keyboard

No.	Time	Source	Destina	Length	Info	Leftover Capture Data
78	37.599…	4.3.1	host	72	URB_INTERRUPT in	0000000000000000
79	37.599…	host	4.3.1	64	URB_INTERRUPT in	
80	46.966…	4.3.1	host	72	URB_INTERRUPT in	0000040000000000
81	46.966…	host	4.3.1	64	URB_INTERRUPT in	
82	47.086…	4.3.1	host	72	URB_INTERRUPT in	0000000000000000
83	47.086…	host	4.3.1	64	URB_INTERRUPT in	
84	47.438…	4.3.1	host	72	URB_INTERRUPT in	0000050000000000
85	47.438…	host	4.3.1	64	URB_INTERRUPT in	
86	47.534…	4.3.1	host	72	URB_INTERRUPT in	0000000000000000
87	47.534…	host	4.3.1	64	URB_INTERRUPT in	
88	47.766…	4.3.1	host	72	URB_INTERRUPT in	0000040000000000
89	47.766…	host	4.3.1	64	URB_INTERRUPT in	
90	47.894…	4.3.1	host	72	URB_INTERRUPT in	0000000000000000
91	47.894…	host	4.3.1	64	URB_INTERRUPT in	

In [7, p.53] the byte values 04 and 05 stand for the a and b keys respectively.

References

USB standard

[1] USB Complete: Everything You Need to Develop Custom USB Peripherals (3rd edition). Jan Axelson, Lakeview Research LLC 2012. ISBN: 978-1-931448-03-1

USB sniffing

[2] The usbmon: USB monitoring framework. Pete Zaitcev, Proceedings of the Linux Symposium, Volume Two, July 20-23 2005.

Links

USB standard

[3] Universal Serial Bus Specification, Revision 2.0. April 27, 2000. Available at http://www.usb.org/developers/docs/usb20_docs/

USB sniffing

[4] https://www.totalphase.com/solutions/apps/usb-analyzer-benefits/

[5] https://wiki.wireshark.org/CaptureSetup/USB Explains how to set up Wireshark for USB capture, also describes sniffing of USB traffic from Windows running inside VirtualBox

[6] http://www.usb.org/developers/hidpage/HID1_11.pdf

[7] http://www.usb.org/developers/hidpage/Hut1_12v2.pdf USB HID Usage tables, describes meaning of bytes in HID reports