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

Hugo Platzer

University of Salzburg

December 28, 2017

1 Introduction

Describe the scanner and motivation

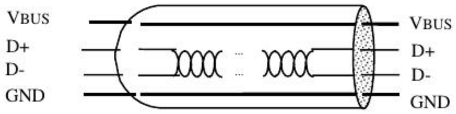
2 The USB standard

2.1 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).

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



2.2 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 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.]

2.3 Signaling

2.3.1 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]

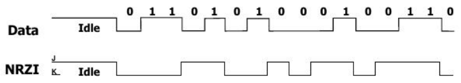
SE0 is the state of the data lines if no device is connected. The host recognizes a device being plugged in by the D+ line being ”pulled up” to high. It then most likely initiates a reset so the device is in a known state for communication to begin. Similarly, a disconnect is sensed by a SE0 for some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

2.3.2 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]

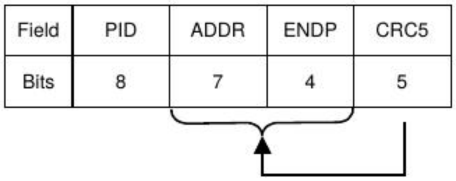
2.4 Packets

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]:

2.4.1 Token packet

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



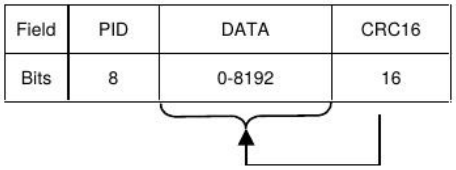
Field	PID	ADDR	ENDP	CRC5
Bits	8	7	4	5

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]

2.4.2 Data packet

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



Field	PID	DATA	CRC16
Bits	8	0-8192	16

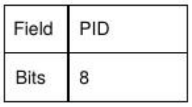
Table 3: USB packet types; notice how the least-significant two bits identify the packet category [3, p. 196]

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

Used to transmit the actual data in a transaction.

2.4.3 Handshake packet

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



Field	PID
Bits	8

Used to report the status of a transaction.

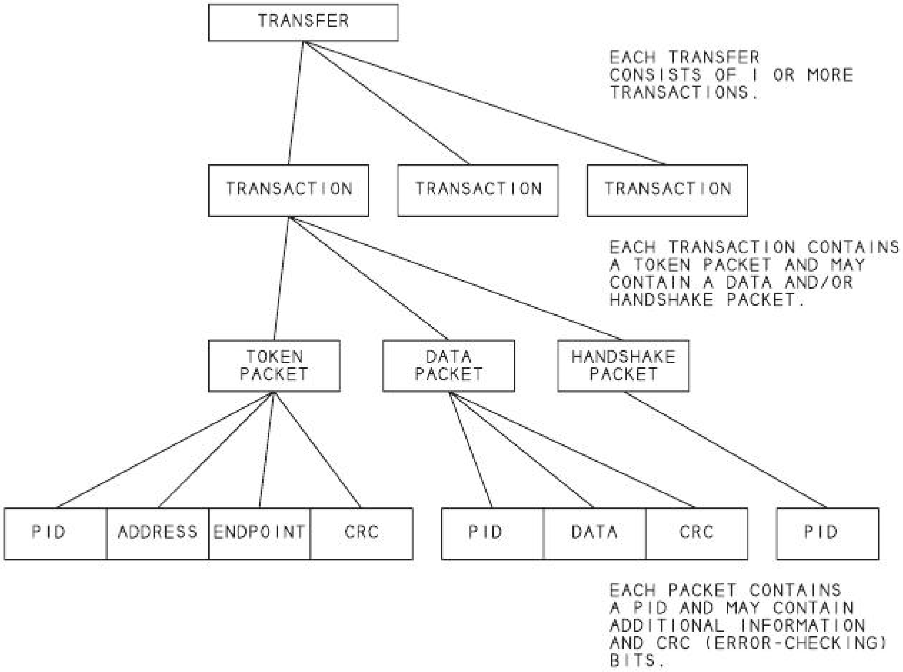
2.5 Transfers

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]


TRANSFER
EACH TRANSFER CONSISTS OF 1 OR MORE TRANSACTIONS.
TRANSACTION
TRANSACTION
TRANSACTION
EACH TRANSACTION CONTAINS A TOKEN PACKET AND MAY CONTAIN A DATA AND/OR HANDSHAKE PACKET.
TOKEN PACKET
DATA PACKET
HANDSHAKE PACKET
PID
ADDRESS
ENDPOINT
CRC
PID
DATA
CRC
PID
EACH PACKET CONTAINS A PID AND MAY CONTAIN ADDITIONAL INFORMATION AND CRC (ERROR-CHECKING) BITS.

2.5.1 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	wLength	Data
GET_DESCRIPTOR	10000000	Descriptor type and index	Zero or Language ID	Descriptor length	Descriptor
SET_ADDRESS	00000000	Device address	Zero	None	None
GET_STATUS	10000000 10000001 10000010	Zero	Zero Interface Endpoint	Two	Device, Interface or endpoint status
SET_CONFIGURATION	00000000	Configuration value	Zero	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.

2.5.2 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.]

2.5.3 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.]

2.5.4 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.]

2.6 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.

2.6.1 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.

2.6.2 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)

2.6.3 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

2.6.4 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)

2.6.5 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

3 USB sniffing

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

3.1 Hardware vs. software 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.

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 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.

Advantages of purely software USB analyzers:

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

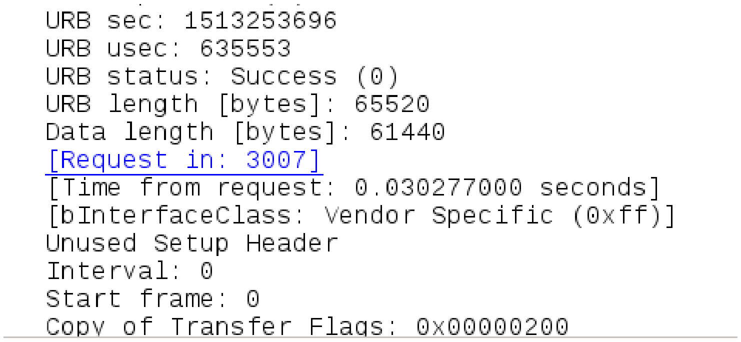
3.2 usbmon

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

3.2.1 Payload size limitation

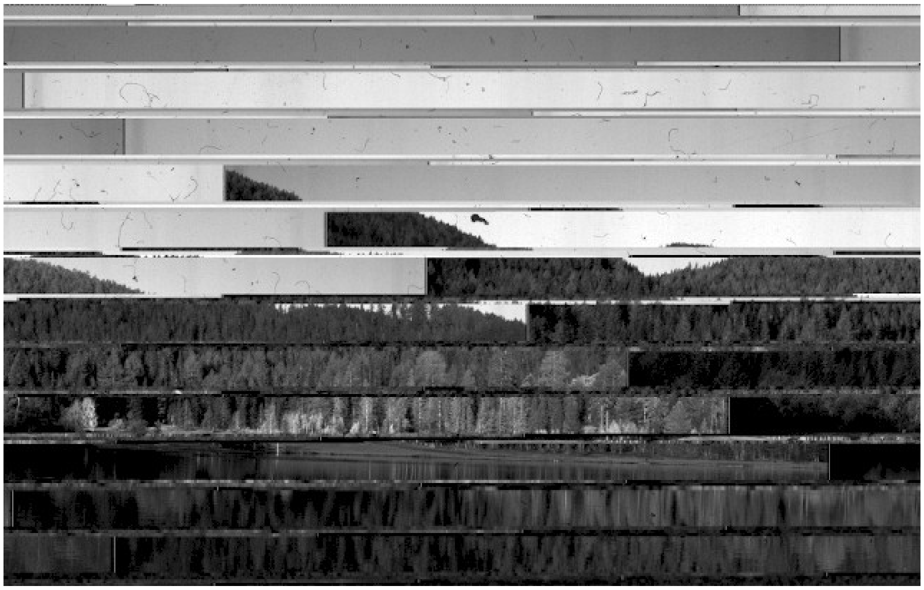
An interesting pitfall when using *usbmon* is that for URBs with a payload longer than 61440 bytes only the first 61440 are captured, the rest is truncated. Others have faced this problem, unsure where it comes from and what to do: [5]

Figure 7: A large bulk transfer in Wireshark, *Data length* < *URB length* indicates missing bytes.



```
URB sec: 1513253696
URB usec: 635553
URB status: Success (0)
URB length [bytes]: 65520
Data length [bytes]: 61440
[Request in: 3007]
[Time from request: 0.030277000 seconds]
[bInterfaceClass: Vendor Specific (0xff)]
Unused Setup Header
Interval: 0
Start frame: 0
Copy of Transfer Flags: 0x00000200
```

Figure 8: Image reconstructed from payloads. The frequent tearing is caused by the large 65520 byte transfers being truncated to 61440 bytes.



3.2.2 Kernel patching

Since the problem persisted when changing computers, sniffing software and target devices it was concluded it must be in the *usbmon* kernel module itself. The *linux-source* package (kernel version 4.13) was installed on the *Debian 9* system.

Source code for *usbmon* is found in *drivers/usb/mon*. *usbmon* has a text-based and a binary interface. The binary interface is used by external sniffing software like Wireshark. The code for it is in *mon_bin.c*.

The magic limit 61440 was not found in the code. However the *mon_bin_event* procedure which is involved in extracting data from a current kernel USB event has an interesting statement:

```
    if (length >= rp->b_size/5) length = rp->b_size/5; .
```

The default buffer size (this value lands in `rp->b_size`) is defined as

```
    #define BUFF_DFL   CHUNK_ALIGN(300*1024) .
```

This gives a maximum length of $\frac{300*1024}{5} = 61440$. Data payloads less than one fifth of the buffer size are truncated for unknown reasons. This line was changed to remove the limitation:

```
    if (length >= rp->b_size) length = rp->b_size; .
```

The modified kernel was compiled and installed according to standard Debian procedures [6]. When running this kernel, the full 65520 bytes of payload could be captured. No adverse side effects were noticed.

3.3 Wireshark

Wireshark [7] is a cross-platform sniffing and packet analysis software that became famous as an Ethernet (network) traffic sniffer. It can be used both for recording traffic as well as analyzing it. For analyzing, it comes with many protocol dissectors that subsequently decode the layers of the packet giving a readable description of the protocol fields' values and their meaning.

Also part of Wireshark is the *tshark* [8] utility. It is basically a command-line version of Wireshark, also accessing the same configuration files, this means behavior of *tshark* can be configured inside Wireshark, for instance which protocols are enabled. *tshark* is ideal for scripts that need to parse the *.pcapng* capture files generated by Wireshark. Packets can be filtered by field values and only the desired fields printed.

Wireshark version 2.2.6 was used, other versions may be significantly different.

3.3.1 Steps

1. Find the bus the device is on using the *lsusb* command. Ideally, make sure there are no other devices on this bus (except the root hub). Try to plug it into different ports.

2. Load the *usbmon* kernel module. (*modprobe usbmon* as root)

3. Launch *wireshark* as root.
4. Select the *usbmon** input corresponding to the bus found in Step 1 and press Start.
5. When completed, stop the recording and save to disk.

3.3.2 Customize columns

For USB sniffing, it is useful to have the most important packet fields as columns. This allows a good insight into the data flow at a certain time when scrolling through the packet list. Also, packets can be sorted using some column, allowing them to be grouped according to endpoint, payload length etc.

To manage columns, go to *Edit* → *Preferences* → *Columns*. New columns are best added by right clicking on the desired field in the analysis window and selecting "Apply as column".

A good selection of columns could be: Packet number, Time, Source, Destination, Length, Info, Leftover capture data (*usb.capdata*). For going through long sequences of control transfers, columns for the control transfer parameters (*bmRequestType, bRequest, wValue, wIndex, Data fragment*) are helpful.

3.3.3 Disable non-USB protocols

For USB sniffing, none of the supplied protocol dissectors except USB are helpful. Even worse, they can get in the way by seeing the first bytes of the data payload match some pattern which causes them to be interpreted as some protocol while they are just raw pixel values. For these packets, the *usb.capdata* field becomes unavailable which makes their payload ignored when parsing the capture file. This leads to missing image bytes, noticeable as a tear in the image (similar to a truncated payload).

To fix this, go to Analyze → Enabled protocols, click "Disable all" and then enable only USB. [9, Section 10.4]

3.4 Example

3.4.1 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.

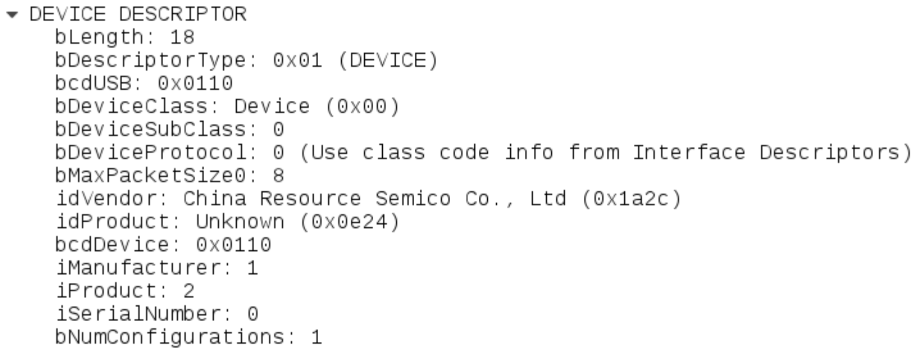
A bus with no devices on it (except the mandatory root hub with address 1) is chosen for sniffing. Recording is started before the device gets attached. The keyboard is attached, then the following keys are pressed and then released (one after another): a, b, a. Recording is then stopped.

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

3.4.2 Device configuration

When the device is connected, a process like in subsection 2.6 starts. The device is reset, an address issued and descriptors queried. The device descriptor and the interface / endpoint descriptor are shown here.

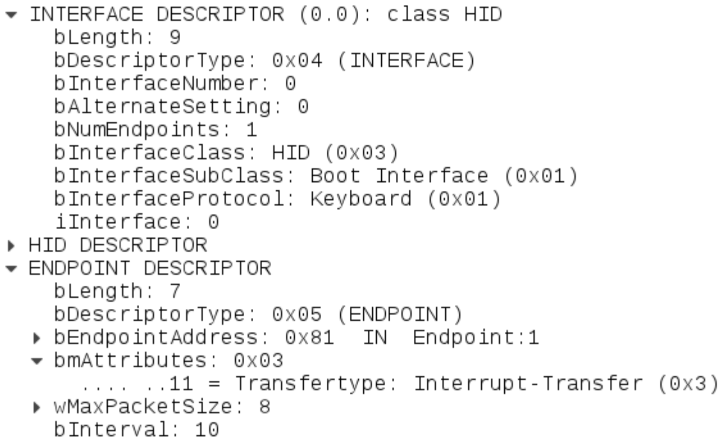
Figure 9: Device descriptor of USB keyboard



```
▾ 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 10: Interface / endpoint descriptor of USB keyboard

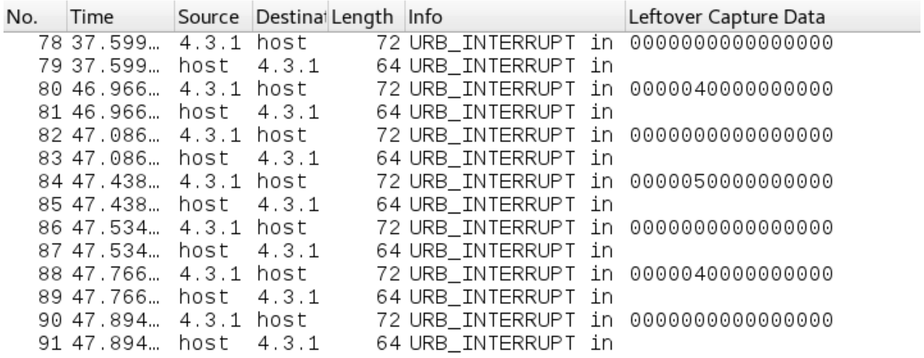


```
▾ INTERFACE DESCRIPTOR (0.0): class HID
    bLength: 9
    bDescriptorType: 0x04 (INTERFACE)
    bInterfaceNumber: 0
    bAlternateSetting: 0
    bNumEndpoints: 1
    bInterfaceClass: HID (0x03)
    bInterfaceSubClass: Boot Interface (0x01)
    bInterfaceProtocol: Keyboard (0x01)
    iInterface: 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. [10]

3.4.3 Data transmitted while keys are pressed

Figure 11: HID reports from USB keyboard



No.	Time	Source	Destinat	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 [11, p.53] the byte values 04 and 05 stand for the a and b keys respectively.

3.5 Sniffing using VirtualBox

When trying to capture device communication from some operating system, it is convenient to take Linux as a host due to its good sniffing capabilities and install the target operating system and device driver in a virtual machine. The virtualization software used here is Oracle VirtualBox [12].

To allow the virtual machine to access the device, the ”USB passthrough” feature is used. This makes the device visible to the guest operating system: VirtualBox relays communication between the actual device on the host and the virtual device connected to a virtual USB controller for the guest. The passthrough is limited to devices with a specified vendor / product ID combination, new devices can be added by selecting the machine and clicking (Settings → USB).

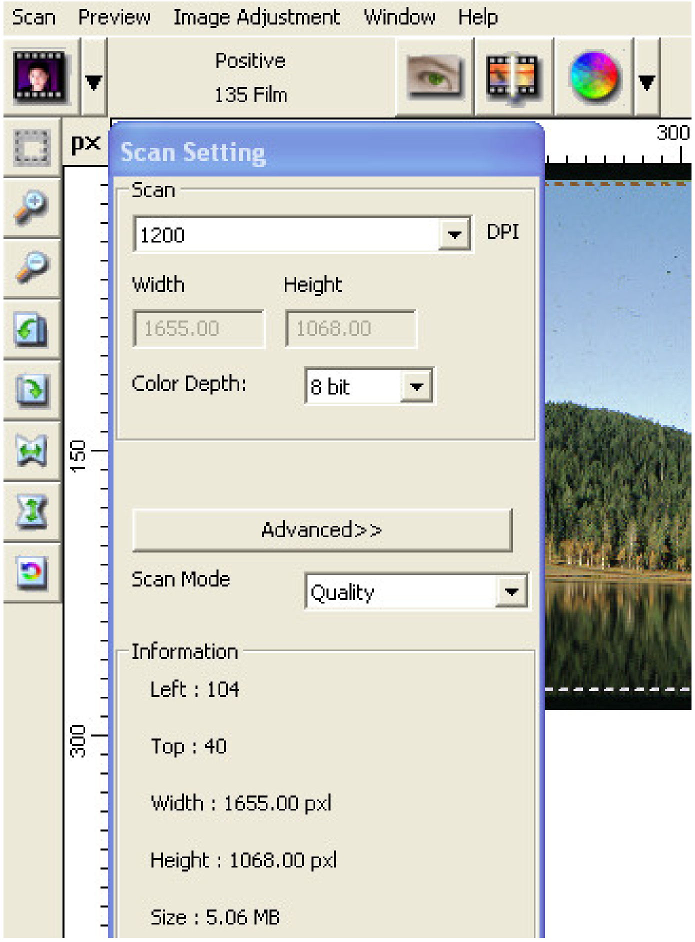
4 Image extraction

As a first step, I wanted to reconstruct the transmitted image from the sniffed communication during a scan. This is done without interaction between my own software and the scanner, only the capture file is analyzed. Extracting the image is also a necessary step when implementing actual scanning software.

4.1 Setup

The Windows XP operating system was set up inside VirtualBox. The vendor-supplied CyberView X5 scanner software [13] (version 5.16) was installed. The scanner was connected to the VM via USB passthrough.

Figure 12: CyberView X5 software



Positive
135 Film

Scan

1200 DPI

Width 1655.00

Height 1068.00

Color Depth: 8 bit

Scan Mode Quality

Information

Left : 104

Top : 40

Width : 1655.00 pxl

Height : 1068.00 pxl

Size : 5.06 MB

The scanner software was configured to scan a color positive at 1200 dpi resolution. Digital ICE dust correction was enabled (Scan → Preference → Positive → ICE / ROC / GEM) to make the scanner record the infrared channel as well.

The scan (Scan → Scan) was performed without doing any pre-scans so as to get any kind of calibration data being exchanged only during the first scan.

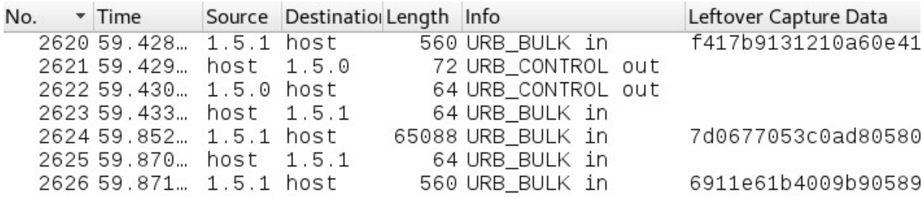
Wireshark recording was started before starting the scanner software and stopped after the software had saved the image.

It is important to use a kernel capable of recording the full payload (subsubsection 3.2.2) and also to disable extra protocols in Wireshark (subsubsection 3.3.3).

A color slide photo of a lake, mountain peaks, green / yellow trees, reflections in the water and sky was used as target. It allows one to easily know which channel (when having them as separate greyscale images) is red, green or blue and provides a good impression of overall image quality.

4.2 Looking at the capture

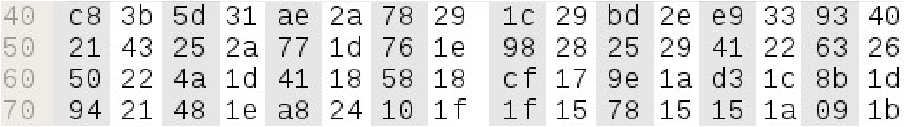
Figure 13: Large bulk transfers



No.	Time	Source	Destinatio	Length	Info	Leftover Capture Data
2620	59.428…	1.5.1	host	560	URB_BULK in	f417b9131210a60e41
2621	59.429…	host	1.5.0	72	URB_CONTROL out	
2622	59.430…	1.5.0	host	64	URB_CONTROL out	
2623	59.433…	host	1.5.1	64	URB_BULK in	
2624	59.852…	1.5.1	host	65088	URB_BULK in	7d0677053c0ad80580
2625	59.870…	host	1.5.1	64	URB_BULK in	
2626	59.871…	1.5.1	host	560	URB_BULK in	6911e61b4009b90589

The scan took about 30 seconds and generated a capture around 18 megabytes in size. The image saved by the scanner software was 1644 x 1069 pixels large. When scrolling through the capture, there are many large bulk transfers incoming from endpoint 1. It seems likely these contain the image data. Since scanners work on a line-by-line basis and the data is transferred in small chunks during the scan, the pixels are probably transferred one line after the other.

Figure 14: Even (dark) and odd (light) offset bytes of payload



```
40  c8 3b 5d 31 ae 2a 78 29  1c 29 bd 2e e9 33 93 40
50  21 43 25 2a 77 1d 76 1e  98 28 25 29 41 22 63 26
60  50 22 4a 1d 41 18 58 18  cf 17 9e 1a d3 1c 8b 1d
70  94 21 48 1e a8 24 10 1f  1f 15 78 15 15 1a 09 1b
```

When looking at the payload of one of the transfers, bytes with odd offset have little variation among their neighbors while those with even offset seem almost random. This suggests the scanner delivers uncompressed 16 bit per pixel little-endian data: The odd bytes contain the approximate brightness and the even bytes the fine nuances (noise).

4.3 Extraction tool

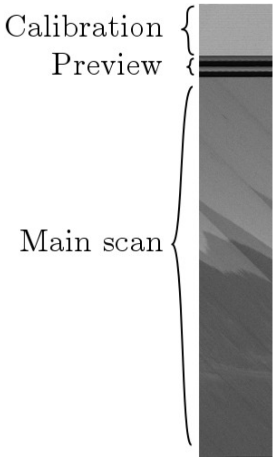
Since we are likely dealing with uncompressed line-by-line image data, it should be possible to construct the image by concatenating the payload bytes and partitioning them into rows. For this, a Python script *extractImage.py* was created, it performs the following steps:

1. Load the *.pcapng* file generated by Wireshark using *tshark* (option *–inputFile*). Make *tshark* print the payload field (*usb.capdata*) as hexadecimal for all packets on endpoint 1. Parse and concatenate the lines of *tshark*'s output into one string of bytes.

2. Apply an offset to the bytes and only keep every n-th (options *–byteOffset*, *–byteNth*). This removes unwanted data at the beginning and only keeps the high byte of the 2 bytes per pixel.
3. Break up the byte string into same-size chunks of a specified size (option *–lineLength*), each representing a line of the image.
4. Apply an offset to the lines and only keep every n-th (options *–lineOffset*, *–lineNth*).
5. Drop all lines beyond a specified maximum (option *–lineMax*). This removes abnormal lines at the end and brings all color channels to the same dimensions.
6. Create a grayscale PNG image, dimensions are the line width and the number of remaining lines. Write the PNG file (option *–outputFile*).

4.4 Results

Figure 15: Transferred bytes, line width 1644



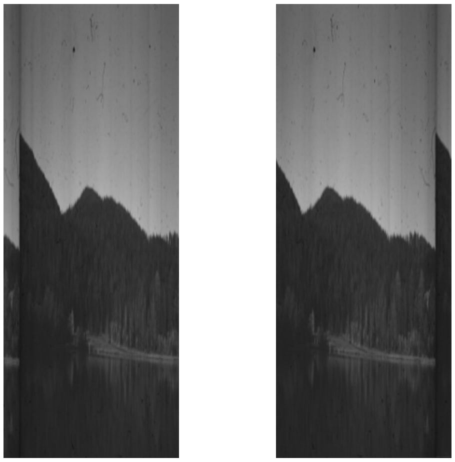
The transferred bytes form three distinct images: The first section is a calibration scan, the second is a low-resolution preview scan and the third (largest) is the actual 1200 dpi scan. The main scan will be analyzed next, while the calibration scan is analyzed in subsection 4.5. The pre-scan is not studied further since its purpose can be served by doing a low-dpi regular scan.

4.4.1 Offsets, line width

To find the exact line width, the guessed line width was incremented / decremented by trial-and-error. The line width is correct when vertical lines in the image (mountain peaks in this case) no longer appear distorted. The correct line width was found to be 1645.

Only the high bytes were kept to transform the 16-bit pixel values to 8-bit grayscale Figure 4.2, this is done with the *–byteNth 2* option and a proper *–byteOffset*. In this capture, the high bytes are at even offsets (use trial-and-error), so *–byteOffset* must be even to keep high rather than low bytes. This however depends on the number of bytes before the actual image.

The byte offset is then further adjusted (starting at zero, in steps of two) to synchronize the start of a line in the recreated image with the start of a line in the scanned image. This can be done quickly using bisection:



(a) offset too low: 0

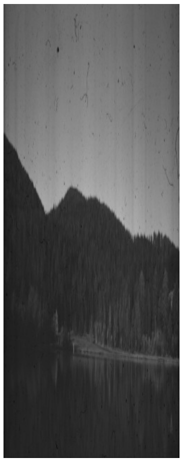
(b) offset too high: 600

Figure 16: Bisecting the correct byte offset

If the offset is slightly too low, the rightmost part of the image will be moved to the very left. If it is slightly too high, the leftmost part of the image will be moved to the very right. An offset of 300 bytes was found to be ideal.

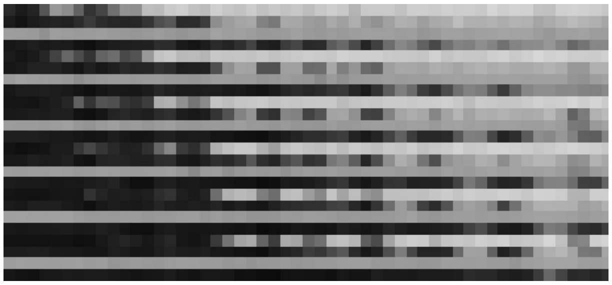
The line offset was adjusted to remove lines not belonging to the main scan, this can be inspected in an image editor such as GIMP [14]. A line offset of 840 was found to look good, removing unwanted lines on top while keeping all of the main scan.

Figure 17: Main scan after adjusting *byteOffset, byteNth, lineOffset*

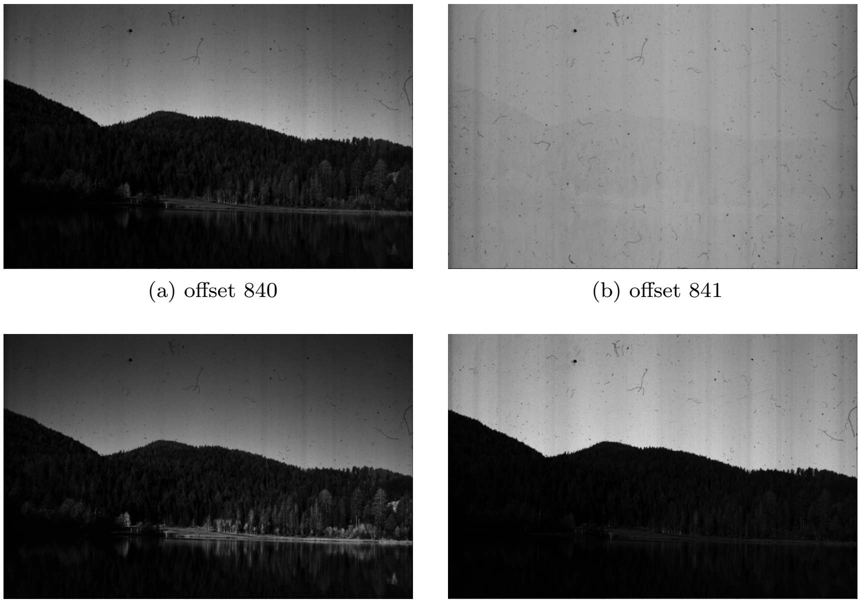


4.4.2 Channel deinterleaving

Figure 18: Interleaved color channels



When looking at the image of Figure 17 in detail, the pixels follow an interleaved line pattern of length 4. For each of the different channels red, green, blue and infrared there is a suitable value modulo 4 such that the image of this channel is defined by all lines whose offset is congruent to that value mod 4. This is the purpose of the *–lineNth* option together with a proper *–lineOffset*. *–lineNth* is set to 4 and then the previously determined line offset of 840 is incremented by 0, 1, 2 and 3 to get four separate color channel images:



(a) offset 840

(b) offset 841

(c) offset 842

(d) offset 843

Figure 19: Four different line offsets / color channels

Thanks to the choice of image scanned, knowing which offset corresponds to what color channel is easy: 840 is green, 841 infrared (only dust visible), 842 red (bright yellow trees), 843 blue (bright sky). The order of channels can vary depending on how much data was skipped at the beginning.

The infrared channel will be used in section 6 to remove dust at the other color channels.

4.4.3 Channel alignment, combination, gamma correction

The extracted color channels are vertically misaligned, to fix this the channel which is the furthest "up" (i.e. the channel for which all other channels only need upwards shifting to align) is determined. Then, for each of the other channels the number of lines it needs to be moved up to align is determined. This is done visually in GIMP by opening the channels as layers and reducing the opacity of the channel to be aligned. The dust specks on the scan are ideal anchors for alignment.

The red channel was found to be the furthest up, the green and infrared channels needed to be moved up 3 lines and the blue channel 5 lines. These numbers are multiplied by 4 (because of interleaving) and added to the channel line offsets found in subsubsection 4.4.2. The line offsets now are: red 842 (same as before), green 852 (+12), infrared 853 (+12), blue 863 (+20).

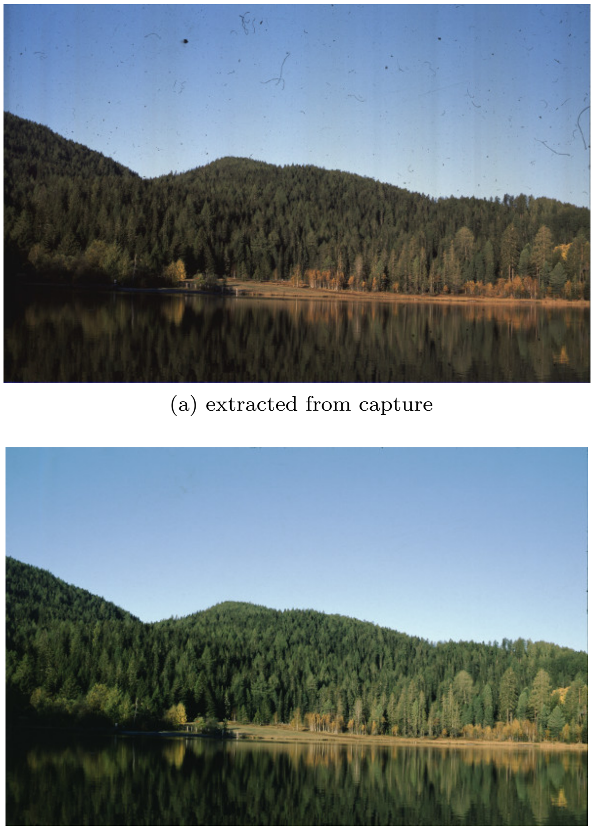
Due to the different line offsets, some channels have more lines than others.

For combining channels, all need to be the same size. To achieve this, a maximum of 1060 lines was kept per channel. This leads to these final parameters:

Table 11: Final parameters per channel

channel	byteOffset	byteNth	lineLength	lineOffset	lineNth	lineMax
red	300	2	1645	842	4	1060
green	300	2	1645	852	4	1060
blue	300	2	1645	863	4	1060
infrared	300	2	1645	853	4	1060

The red, green and blue channels are combined into an RGB image. To make the image look a little more presentable, a gamma correction with $\gamma = 2$ was applied on each channel. This means the brightness values get normalized to the range $[0, 1]$, then a mapping $x \mapsto x^{\frac{1}{2}}$ is applied, then the values are converted back to the range $[0, 255]$. This is the resulting 1045x1060 color image:



(a) extracted from capture

(b) saved by CyberView X5

Figure 20: Extracted image compared to the scanner software

The color balance / saturation / contrast is different to the original because of postprocessing done by CyberView. Vertical stripes on the extracted image could be removed with the help of the calibration scan. CyberView also removes dust using the infrared channel (called Digital ICE).

4.5 Calibration image

5 Controlling the device

5.1 Setup: PyUSB

PyUSB [15] is a Python module for accessing USB devices. Using it is straightforward: A handle for the device is created by specifying the characteristic vendor and product ID (these are for my scanner, can be found via *lsusb*):

```
dev = usb.core.find(idVendor=0x05e3, idProduct=0x0145)
```

After finding the scanner, like any USB device it needs to be configured before using it. This means choosing one of the available configuration profiles (described by configuration descriptors). The scanner, like most devices, only has one configuration. It can be selected with:

```
dev.set_configuration(0)
```

After configuring the device, control and bulk transfers can be initiated. Control transfers are initiated like this:

```
dev.ctrl_transfer(self, bmRequestType, bRequest, wValue, wIndex,
                   data_or_wLength, timeout)
```

bmRequestType, bRequest, wValue, wIndex, wLength are described in Table 4. timeout specifies the number of milliseconds after which the transfer is cancelled if still not completed.

For host-to-device control transfers, *data_or_wLength* contains the data payload and the return value is the number of bytes actually written. The wLength parameter is inferred from the length of the payload bytestring.

For device-to-host control transfers, *data_or_wLength* contains the wLength parameter specifying the number of bytes to be read and the return value is the payload received from the device.

As opposed to control transfers, bulk transfers have no structure at the layer of the USB standard and can be imagined as reading from / writing to a byte stream. They are self-explanatory:

```
dev.read(endpoint, size, timeout)
dev.write(endpoint, data, timeout)
```

5.2 Vendor-specific control transfers

USB traffic starts when the scanner software loads and continues constantly while the software is running (even when idling). It consists of vendor-specific control transfers on endpoint 0 in both directions: bmRequestType 0x40 outgoing / 0xc0 incoming, see Table 4. Also there are incoming bulk transfers on endpoint 1, used both for receiving status information and image data.

These combinations of control transfer parameters were observed while the software was idling / scanning:

Table 12: Control transfer parameter combinations

bmRequestType	bRequest	wValue	wIndex	wLength
0x40	4	0x0082	0	8
0xc0	12	0x0084	0	1
0x40	12	0x0085	0	1
0x40	12	0x0087	0	1
0x40	12	0x0088	0	1

Some additional parameter combinations were only observed during software startup:

Table 13: Parameter combinations at startup

bmRequestType	bRequest	wValue	wIndex	wLength
0x40	12	0x0089	0xf49c	1
0xc0	12	0x008a	0x008b	1
0x40	12	0x008b	varies	1
0x40	12	0x008c	0x0010	1
0xc0	12	0x008e	0	1

The data payload sent / received varies between different transfers with the same parameters.

5.3 Initiating a scan

5.4 Receiving the image

6 Image processing

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] http://permalink.gmane.org/gmane.linux.usb.general/101338 Others describing issues with usbmon truncating payloads

[6] https://www.debian.org/releases/stretch/amd64/ch08s06.html.en

[7] https://www.wireshark.org/ Homepage of the Wireshark network protocol analyzer

[8] https://www.wireshark.org/docs/man-pages/tshark.html

[9] https://www.wireshark.org/docs/wsug_html Wireshark User's Guide

[10] http://www.usb.org/developers/hidpage/HID1_11.pdf USB HID Device Class Definition. Describes structure of HID reports

[11] http://www.usb.org/developers/hidpage/Hut1_12v2.pdf USB HID Usage Tables. Describes meaning of bytes in HID reports

[12] https://www.virtualbox.org/

[13] https://reflecta.de/de/downloads/drivers2/~nm.50~nc.108/Treiber-und-Software.html

[14] https://www.gimp.org/

[15] https://github.com/pyusb/pyusb Control USB devices from Python. Also has a nice tutorial