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

Hugo Platzer

University of Salzburg, Austria

April 12, 2018

1 Introduction

For many vendors of consumer electronic devices, developing drivers for desktop operating systems with a small market share (such as Linux distributions) is not a priority. Even if they do develop them, their software usually remains closed-source, which can be considered undesirable from a security and maintainability standpoint. And even for those vendors releasing open-source drivers, the device's communication protocol generally remains undocumented.

This paper focuses on a scanner for film material. The communication between the device and the supplied Windows software is recorded and analyzed. Based on this analysis, a working driver supporting basic scanning functionality is created.

1.1 The device

The Reflecta CrystalScan 7200 is a device for digitizing 35mm film material like slide, color negative and black-and-white negative film. It uses a linear CCD sensor on a head moving vertically to record the image on a row-by-row basis. The maximum scan resolution is 7200 dots per inch which amounts to around 70 megapixels on a 35mm film frame. It also has a feature called Digital ICE: In addition to red, green, blue, it also records an infrared channel of the film. The dyes in color film are transparent to infrared light, but dust is not, so the information from this channel is useful for removing it in software. This scanner is connected via USB. [3] [4]

1.2 Software support

Reflecta supplies a software called CyberView X5 with the scanner that provides basic functionality (adjust resolution, scan area, image processing parameters, dust correction). The software only runs on the Windows and macOS operating systems.

Figure 1: Reflecta CrystalScan 7200

For the Linux operating system, there is a community effort to build scanning software called SANE [20]. As opposed to vendor-supplied software, it supports many scanners from various manufacturers. Unfortunately, my scanner was not yet supported. There exist third-party software products that support this scanner: VueScan [5] and SilverFast [6]. VueScan also runs on Linux. These however are commercial software and not open-sourced.

1.3 Reverse engineering

Since there is no publicly available documentation on how a driver should control this device, reverse engineering was attempted. Reverse engineering is the process of reconstructing a specification from a finished product. Typically, the finished product is software or some sort of electronic appliance. Using only resources available to the end user (like hardware, software / firmware in machine code form, communication channels) one tries to gain understanding about the inner mechanisms of the product.

In my case, I recorded and analyzed the activity on the USB bus while the scanner was communicating with the vendor-supplied software.

2 The USB standard

2.1 Motivation

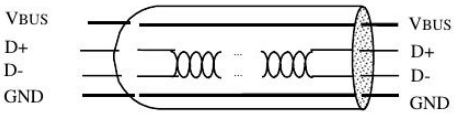
USB is an interface intended to connect various peripherals to PCs. These include: Human interface devices like keyboards and mice; storage devices like card readers, external hard disks, memory sticks and smartphones; multimedia devices like microphones, speakers, cameras and scanners. Some highlights leading to its wide adoption [7, 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).

2.2 Electrical side

Figure 2: USB cable cross-section [7, p. 17]



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. [7, 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 [7, 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) [7, 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. [7, p. 12]

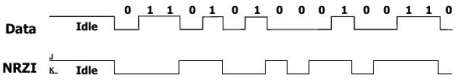
SE0 is the state of the data lines if no device is connected. The host recognizes a device being plugged in by the D+ line being "pulled up" to high. It then most likely initiates a reset so the device is in a known state for communication to begin. Similarly, a disconnect is sensed by a SE0 for some time. [7, 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. [7, 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 3: NRZI bitstream encoding [7, 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. [7, p. 157]

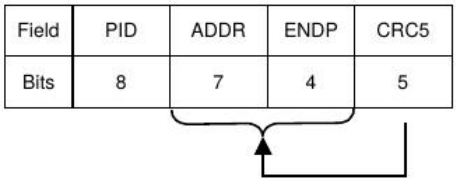
2.4 Packets

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. [7, 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) [7, p. 195]:

2.4.1 Token packet

Figure 4: Token packet format [7, 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). [7, 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. [7, p. 33]

2.4.2 Data packet

Figure 5: Data packet format [7, p. 206]

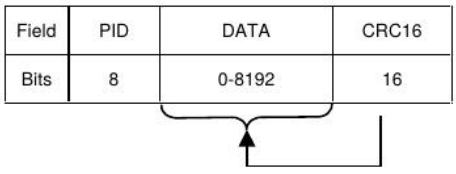


Table 3: USB packet types; notice how the least-significant two bits identify the packet category [7, p. 196]

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 6: Handshake packet format [7, p. 206]

Field	PID
Bits	8

Used to report the status of a transaction.

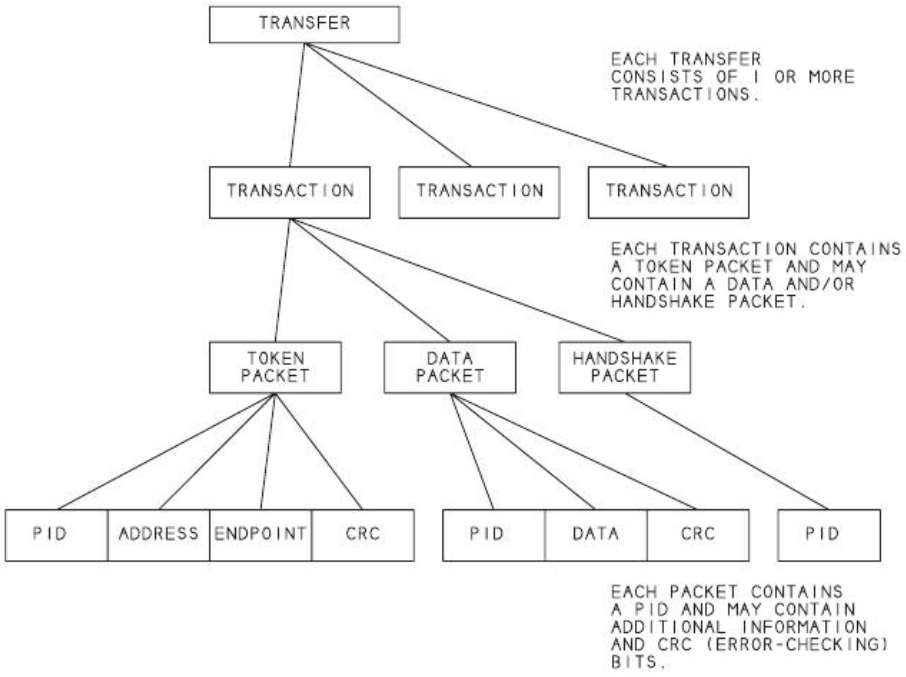
2.5 Transfers

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 [7, 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 7: Composition of a USB transfer [1, p. 44]



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. [7, p. 38f.]

Table 4: Device Request format [7, 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 [7, 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.

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. [7, 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. [7, 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. [7, 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 [7, 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. [7, p. 244]

Table 7: Configuration descriptor format [7, 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. [7, p. 244]

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

Table 8: Interface descriptor format [7, 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 [7, 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 [7, 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 [10] :

- 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]

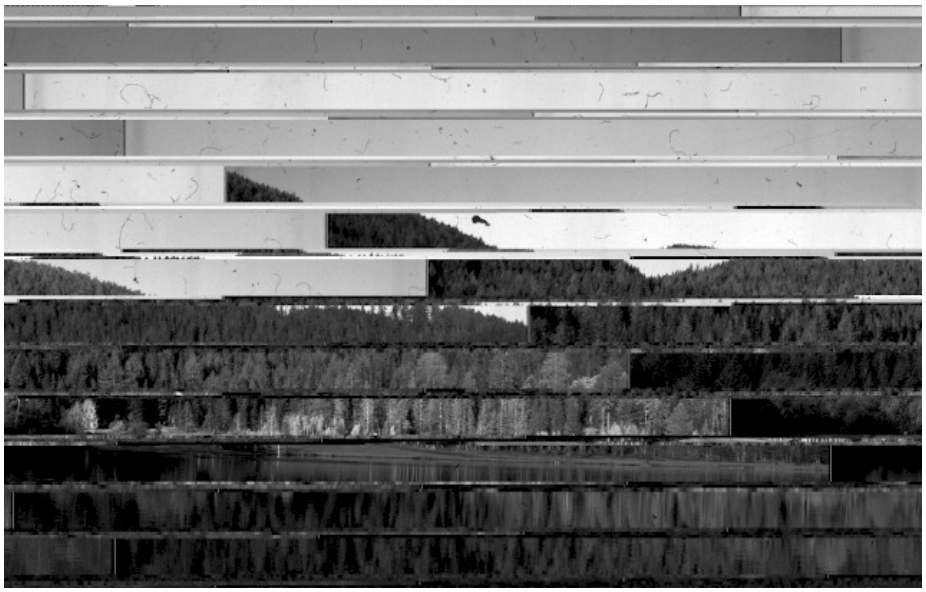
3.2.1 Payload size limitation

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: [11]

Figure 8: 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 9: 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 [12]. When running this kernel, the full 65520 bytes of payload could be captured. No adverse side effects were noticed.

3.3 Wireshark

Wireshark [13] 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* [14] 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. [15, 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 10: 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 11: 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. [8]

3.4.3 Data transmitted while keys are pressed

Figure 12: 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 [9, 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 [16].

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).

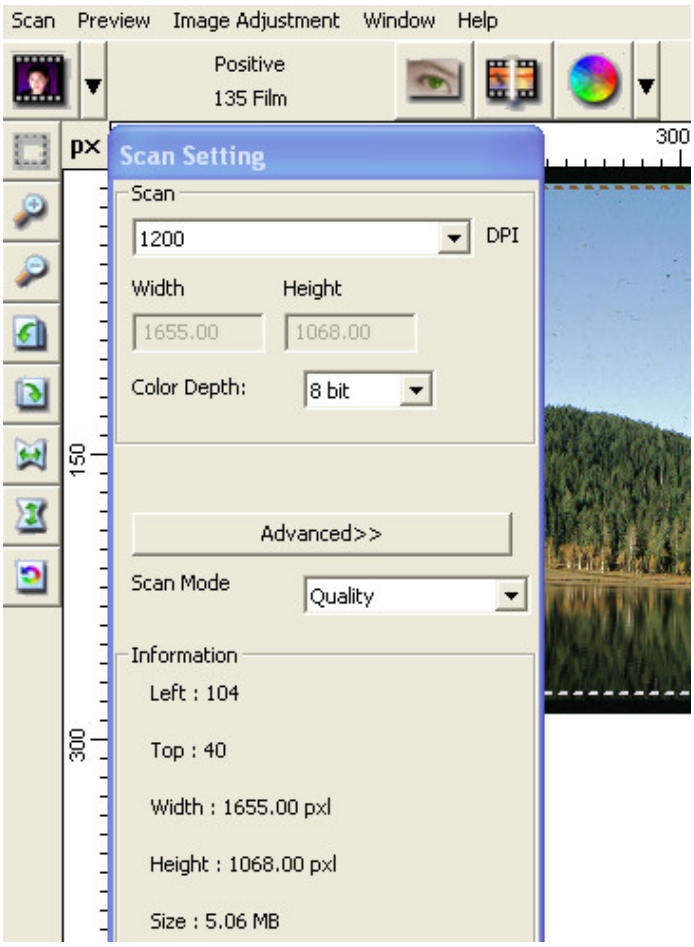
4 Image extraction

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 [17] (version 5.16) was installed. The scanner was connected to the VM via USB passthrough.

Figure 13: CyberView X5 software



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 14: 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 15: 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.

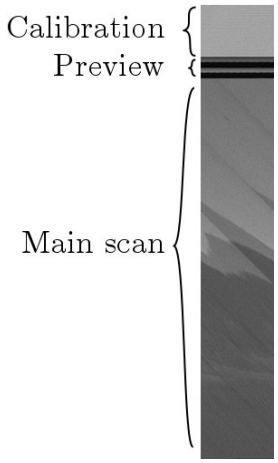
4. Apply an offset to the lines and only keep every n-th (options *–lineOffset*, *–lineNth*).

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 16: 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.

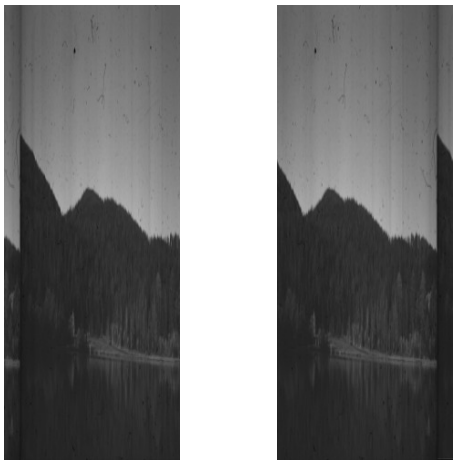
4.4.1 Offsets, line width

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 17: Bisecting the correct byte offset

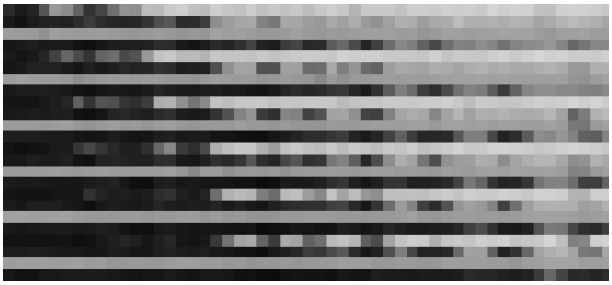
If the offset is slightly too low, the rightmost part of the image will be moved to the very left. If it is slightly too high, the leftmost part of the image will be moved to the very right. An offset of 300 bytes was found to be ideal.

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 [18]. A line offset of 840 was found to look good, removing unwanted lines on top while keeping all of the main scan.

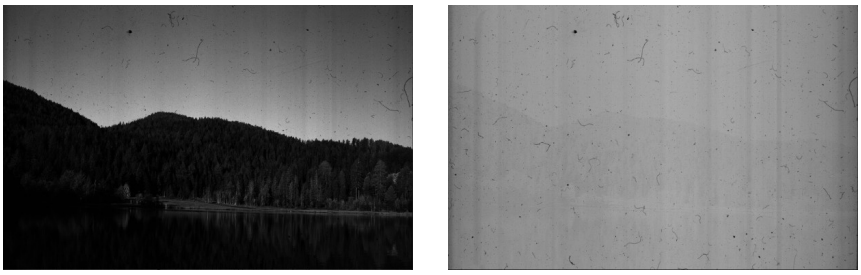
Figure 18: Main scan after adjusting *byteOffset, byteNth, lineOffset*

4.4.2 Channel deinterleaving

Figure 19: Interleaved color channels



When looking at the image of Figure 18 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 20: 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 7 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 21: 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 [19] 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)
```

PyUSB was used to create a simple scanning program *scanInteractive.py* which performs a scan with certain parameters, does basic image processing (see 4) and displays the image building up row by row as data is received from the scanner.

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

Combination	bmRequestType	bRequest	wValue	wIndex	wLength
a)	0x40 (outgoing)	4	0x0082	0	8
b)	0xc0 (incoming)	12	0x0084	0	1
c)	0x40 (outgoing)	12	0x0085	0	1
d)	0x40 (outgoing)	12	0x0087	0	1
e)	0x40 (outgoing)	12	0x0088	0	1

These additional parameter combinations were only observed during software startup:

Table 13: Parameter combinations at startup

Combination	bmRequestType	bRequest	wValue	wIndex	wLength
f)	0x40 (outgoing)	12	0x0089	0xf49c	1
g)	0xc0 (incoming)	12	0x008a	0x008b	1
h)	0x40 (outgoing)	12	0x008b	varies	1
i)	0x40 (outgoing)	12	0x008c	0x0010	1
j)	0xc0 (incoming)	12	0x008e	0	1

In addition to the parameters, control transfers have a data payload that is *wLength* bytes long Table 4. The actual commands regarding what the device should do are encoded in this data payload. As all but one parameter combination has a one-byte payload, most commands are transferred one byte after another, doing one control transfer for each byte.

5.3 Transaction patterns

All traffic during idling and scanning can be modelled by transactions: Each transaction starts with the same header. After that, six parameter bytes are sent and the device response is queried. The rest depends on the transaction type, there are four types of transaction: The most basic does not do anything besides sending parameters and querying the response. One kind of transaction transmits additional parameter bytes, another reads more detailed device status and the fourth kind is used for retrieving image data.

5.3.1 Header

Every transaction begins with the same header that consists of these 11 control transfers:

Table 14: Header structure for all three transaction types

Nr.	Parameters Table 12	Data payload
1.	e)	0xff
2.	e)	0xaa
3.	e)	0x55
4.	e)	0x00
5.	e)	0xff
6.	e)	0x87
7.	e)	0x78
8.	e)	0xe0
9.	d)	0x05
10.	d)	0x04
11.	e)	0xff

5.4 Device readiness query

At the end of each transaction, but also at other points (like before sending extra parameters, before transferring image data) the software has to ask for a device response.

This is done using a control transfer with parameter combination b) in Table 12.

Table 15: Device readiness responses

Response	Meaning
`0x00`	OK, returned inside extra parameter transaction (before sending extra parameters)
`0x01`	OK, returned in status /i mage transaction, before doing bulk reads
`0x03`	Read another response byte, see Table 16

When the first response byte is `0x03`, the control transfer needs to be repeated for another response byte. Such a two-byte response means the transaction is now complete (if in the middle of an image / status transaction, do not continue with this transaction, but repeat the transaction from the start).

Table 16: Second response byte

Response	Meaning
0x00	OK. Continue with next transaction.
0x02	Failure due to wrong commands sent by software. Should not happen during normal operation.
0x08	Device not ready yet. Repeat transaction.

5.5 Reading from the bulk endpoint

For a status transaction or image transaction, data needs to be read from the bulk endpoint of the device. This has to be done using a special pattern:

First, a control transfer (combination a) in Table 12) is performed with the following 8-byte payload (hex): 00000000xxyy0000, where the number of bytes to be read (c, $1 \leq c \leq 65520$) from the bulk endpoint is encoded in little-endian format, i.e. bytes at offset 4 and 5 (xx and yy) are set such that $c = 256 * \texttt{yy} + \texttt{xx}$.

Then the actual bulk read transfer of c bytes from endpoint 1 can happen. The vendor software splits large transfers into multiple smaller transfers, but this does not seem to be neccessary.

5.6 Basic transaction

This type of transaction has the simplest structure. It is used only for a handful of tasks like checking device readiness at critical points and the actual "start scan" command.

5.6.1 Structure

Table 17: Structure of the basic transaction

Stage	No. of transfers / bytes	Transfer parameters	Data payload
Header	11	see Table 14	see Table 14
Main parameters	6	c) in Table 12	see 5.6.2
Device response	2	b) in Table 12	see 5.4

5.6.2 Main parameters

Table 18: Observed values for the "Main parameters" bytestring

Value (hex)	Description
`000000000000`	Used to wait for device readiness at various points while preparing to scan. The transaction is repeated until the device readiness query returns the right response.
`1b0000000100`	The actual "start scan" command. Is issued after transmitting all scan parameters before receiving the image data. After recieving this command, the scanner starts to make noise and eventually moves the scan head.

5.7 Extra parameter transaction

This transaction is used to set parameters while preparing a scan. The bytestring of the "Main parameter" stage defines the purpose of the transaction (e.g. setting resolution / exposure parameters). The "Extra parameter" stage transfers the values of the parameters that should be set.

5.7.1 Structure

Table 19: Structure of the extra parameter transaction

Stage	No. of transfers / bytes	Transfer parameters	Data payload
Header	11	see Table 14	see Table 14
Main parameters	6	c) in Table 12	see 5.7.2
Device response	1	b) in Table 12	see 5.4, `0x00` expected
Extra parameters	varies, see 5.7.2	c) in Table 12	see 5.7.3
Device response	2	b) in Table 12	see 5.4

5.7.2 Main parameters

Only bytes at offset 0 and 4 are set. The byte at offset 0 identifies the purpose of this transaction (e.g. set resolution, scan area). The byte value at offset 4 equals the number of bytes that are later transferred in the "Extra parameter" stage.

Sometimes, the same byte 0 (purpose) is used with varying bytes 4 (length). It is unclear whether such extra parameter bytestrings of different lengths are related to each other. Usually, only one of these lengths is used to transmit a string that actually carries relevant information (i.e. it changes when changing settings in the vendor software).

Table 20: Observed values for the "Main parameters" bytestring

Value (hex)	Description
`0a0000000600`	Unclear purpose
`0a0000000800`	Unclear purpose
`0a0000000e00`	Scan area
`150000001000`	Resolution, color mode
`dc0000001d00`	Exposure / gain

5.7.3 Extra parameters

For three main parameter strings, the extra parameter string varies with different scan settings. The meaning of some bytes in these strings can be discovered by varying one setting and observing the change in the string. Most of the information presented here, however, is taken from comments in the code of the SANE backend for this device. For each of the three parameters, an example string from an actual scan is analyzed:

0a0000000e00 This parameter string is used to set the boundaries of the scan frame / scan area. The corresponding extra parameter bytestring consists of 5 2-byte little-endian values:

Table 21: Decoding extra parameters for "0a0000000e00"

Offset	Value (hex)	Description
0-1	`1200`	Constant representing "scan frame"
2-3	`0a00`	Size of rest of frame (10 bytes), constant
4-5	`8000`	Index of frame, constant
6-7	`6f02`	X coordinate of top left corner of area (units unknown)
8-9	`0f00`	Y coordinate of top left corner of area (units unknown)
10-11	`2829`	X coordinate of bottom right corner of area (units unknown)
12-13	`f819`	Y coordinate of bottom right corner of area (units unknown)

Information taken from comments in this C code: [24, `pieusb_scancmd.c`]

The ratio of x distance to y distance in this example is:
$\frac{0x2928-0x026f}{0x19f8-0x000f} = \frac{9913}{6633} \approx 1.494$.
This matches the proportions of a 35mm film frame: $\frac{36mm}{24mm} = 1.5$.

150000001000 This is used to set many different scan parameters such as resolution, color mode, color depth etc. Multi-byte values are in little-endian format.

Table 22: Decoding extra parameters for "150000001000"

Offset	Value (hex)	Description
0-1	`000f`	Constant representing "scan parameters"
2-3	`b004`	Scan resolution (in dots per inch)
4	`80`	Color mode (`80` - RGB, `90` - RGB + infrared)
5	`80`	Color depth (`04` - 8 bit, `20` - 16 bit)
6	`04`	Color format
7	`00`	unused
8	`01`	Byte order of pixel value (`01` - little endian)
9	`00`	Bitmap for various quality settings (`80` - fast infrared, `08` - skip calibration, `02` - sharpen)
10-11	`0000`	unused
12	`01`	halftone pattern
13	`80`	line threshold
14-15	`1000`	unknown

Information taken from comments in this C code: [24, `pieusb_scancmd.c`]

dc0000001d00 This is used to set some parameters such as exposure time per channel and digitizer gain / offset. Multi-byte values are in little-endian format.

Table 23: Decoding extra parameters for "dc0000001d00"

Offset	Value (hex)	Description
0-5	`7e26661c1515`	exposure time for red, green, blue channel (one 16-bit little-endian value per channel)
6-8	`171411`	Digitizer offset for red, green, blue channel (one 8-bit value per channel)
9-11	`000000`	unused
12-14	`212121`	Digitizer gain for red, green, blue channel (one 8-bit value per channel)
15	`07`	light
16	`00`	extra entries
17	`00`	double times
18-19	`790b`	Exposure time for infrared channel
20-21	`1400`	Digitizer offset for infrared channel
22	`0f`	Digitizer gain for infrared channel
23-28	`000000000000`	unused

Information taken from comments in this C code: [24, `pieusb_scancmd.c`]

5.8 Status transaction

This is used to read device status from the bulk endpoint.

5.8.1 Structure

Table 24: Structure of the read transaction

Stage	No. of transfers / bytes	Transfer parameters	Data payload
Header	11	see Table 14	see Table 14
Main parameters	6	c) in Table 12	see 5.8.2
Device response	1	b) in Table 12	see 5.4, `0x01` expected
Bulk read	see 5.8.2	see 5.5	see 5.5
Device response	2	b) in Table 12	see 5.4

5.8.2 Main parameters

Many different main parameter combinations are used for status reads. As most of the returned information is not critical for basic scanning functionality, not all bytestrings are studied in detail.

Bytes at offset 0, 3, 4 can be set. Bytes 3 and 4 specify the number of bytes to be read from the bulk endpoint: $256 * b_3 + b_4$.

Table 25: Observed values for the "Main parameters" bytestring

Value (hex)	Description
030000000e00	Unknown purpose
080000008000	Unknown purpose
0f0000001200	Main image parameters
18000014dc00	Mask specifying which pixels of the sensor contribute to the output data (useful for calibration)
d70000006700	Gain / offset values
dd0000001200	Basic scanner status. Sent in regular intervals by the vendor software while the scanner is idle.

0f0000001200 This parameter string is used to receive information about the image produced by the scanner. It contains the image width / height, this is needed to use the proper image transaction parameters. Multi-byte values are in little-endian format.

Table 26: Decoding status response for "0f0000001200"

Offset	Value (hex)	Description
0-1	7406	Line width (pixels)
2-3	2c04	Number of lines in the image
4-5	e80c	Number of bytes in the image
6	08	Filter offset 1
7	08	Filter offset 2
8-11	00000000	Exposure period
12-13	7266	SCSI transfer rate
14-15	0000	Number of lines in buffer right now
16-17	0000	unused

Information taken from comments in this C code: [24, `pieusb_scancmd.c`]

5.9 Image transaction

The image transaction is used for receiving the calibration and the main image.

5.9.1 Structure

Table 27: Structure of the image transaction

Stage	No. of transfers / bytes	Transfer parameters	Data payload
Header	11	see Table 14	see Table 14
Main parameters	6	c) in Table 12	see 5.9.2
Device response	1	b) in Table 12	see 5.4
Bulk reads	see 5.9.3	see 5.5	see 5.5
Device response	2	b) in Table 12	see 5.4

5.9.2 Main parameters

An image transaction has the parameter string (hex) `08000000xx00`. The byte at offset 4 represents the number of lines of the image that should be received.

An image transaction that transfers for example 14 lines would be identical up to the main parameter stage to an extra parameter transaction with main parameter string `080000000e00`. The device response after the main parameter stage however has to be different (0x00 for extra parameter, 0x01 for image). This shows that the device has to consider context when analyzing the meaning of a transaction's main parameters (`080000000e00` means "set scan area" while preparing the scan but "transfer 14 lines" after the scan has started).

> This is a SCSI READ command (code 0x08). It is distinguished from the other READ commands by the context in which it is issued: see `sanei_pieusb_cmd_start_scan()` ([24, `pieusb_scancmd.c`])

5.9.3 Bulk reads

In this stage, the image data is transferred using one ore more bulk reads (see 5.5).

For this, the amount of bytes per line of the image c_l needs to be known (see Table 26). This is multiplied by the number of lines l to be transferred to get the total amount of bytes to be transferred:

$$c = c_l * l$$

This amount gets divided into 65520 (`0xfff0`) byte chunks plus a possible rest:

$$n = \lfloor \frac{c}{65520} \rfloor$$
$$m = c \mod 65520$$

The data is transferred in n 65520-byte reads followed by one m-byte read (if m is nonzero).

5.10 Receiving the calibration image

The size and format of the calibration image does not depend on the scanner settings. On this scanner, it has 180 (45 per channel) lines of 5341 16-bit pixels (10682 bytes). The total calibration image size is 1922760 bytes.

The lines follow a RGBI channel interleaving (see 4).

In the actual scan (5.13), at first 4 lines get transferred, then 172, then the last 4.

5.11 Receiving the main image

When receiving the main image, it is essential to know how many lines the image has and how many bytes there are per line. These values are used to generate image transactions.

The number of lines l gets divided into 255-line chunks plus a possible rest:

$$n = \lfloor \frac{c}{255} \rfloor$$
$$m = c \mod 255$$

The image is transferred in n 255-line image transactions followed by one m-line image transaction (if m is nonzero). Each mage transactions itself of course needs to be adjusted to its line count and the number of bytes per line.

If the data is read faster than the scanner can produce it, the device readiness response in the middle of the image transaction will return a busy status. In this case, just repeat this image transaction until the scanner is ready.

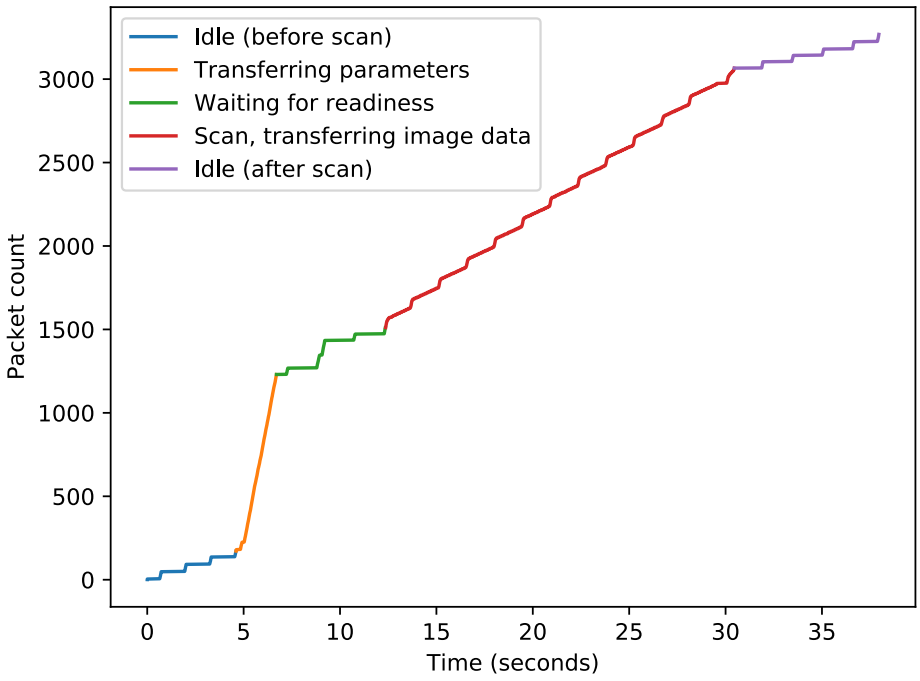
5.12 Timing

Suppose you have a recording of all USB traffic that happened during a scan. It would not be sufficient to just replay all the recorded outgoing packets at once (and wait for incoming packets at the right points).

This is due to the importance of timing. At certain points, the device is not immediately ready for the next command. The most important point where this is the case is after starting the scan before doing the first image transaction. To ensure device readiness, a basic query (5.6) is used. The device response (5.4) in this query returns "busy" if the device is not ready yet. After receiving a busy response, the software should sleep for a bit (the vendor software uses a 1.5 second interval) and repeat the basic transaction until the device responds that it's not busy.

Something similar happens when doing image transactions faster than the device can scan the lines. Before the bulk read stage, the device responds "busy" and the image transaction should be repeated from the start (5.11).

Figure 22: Timing of packets during scan



This diagram illustrates the rate at which packets are transmitted during a simple scan (no prescan, no calibration) with the vendor-supplied (CyberView X5) software. The scanning process can be separated into five phases:

1. During the idle phase before the scan, every 1.5 seconds a status read query (5.8) is performed.

2. When the user presses "start" scan, all scan parameters being transferred in rapid succession causes a quick increase in packet count.

3. After the parameters are sent to the device, but before the image is transferred the software has to wait for readiness (1.5 second waits in between a basic transaction which shows as a burst of packets).

4. During the image transaction stage, each image transaction's header is transferred quickly. The transfer of the actual image lines happens without waits but more slowly (the curve climbs diagonally) due to the large amount of data.

5. After the scan has completed and the image is fully transferred, the traffic returns to the idle pattern again. Interestingly, the delay between status queries is now slightly longer (about 2s) than before the scan.

5.13 An actual scan

All the USB traffic that happens during an actual scan can be represented using the transaction patterns described in this section. To better illustrate how an actual scan proceeds, I describe an example using a list of transactions.

A prescan (includes receiving calibration image) was selected in the vendor software for its calibration image but also because its transactions can be replayed to the device straight after the scanner has started (not possible with a regular scan, the device needs to do at least one pre-scan after having started).

Table 28: Scan parameters

Setting	Value
Scan mode	Pre-scan
Resolution	300 dpi (main image), 3600 dpi (calibration image)
Color mode	RGB (main image), RGBI (calibration image)
Color depth	1 byte per pixel (main image), 2 bytes per pixel (calibration image)

Table 29: Transactions of the scan

Type	Main parameters (hex)	Extra parameters (hex)	Description
S	dd0000001200		Idle status query Table 25
B	000000000000		Readiness check 5.6
E	0a0000000800	1300040002006400	
E	0a0000000800	1300040004006400	
E	0a0000000800	1300040008006400	
E	0a0000000800	1400040002006400	
E	0a0000000800	1400040004006400	
E	0a0000000800	1400040008006400	
E	0a0000000600	950000000000	
S	080000008000		
E	0a0000000e00	12000a0080000000000b829e71a	Set scan area Table 21
E	0a0000000600	170002000100	
S	030000000e00		
B	000000000000		Readiness check 5.6
S	d70000006700		Read gain / offset values 5.8.2
E	dc0000001d00	0000000000000000000000000007001a790b00fe0f000000000000	Set exposure settings Table 23

Transaction types: B - Basic, E - Extra parameter, S - Status, I - Image

Table 30: Transactions of the scan (continued)

Type	Main parameters (hex)	Extra parameters (hex)	Description
B	000000000000		Readiness check 5.6
E	150000001000	000f2c0180 0404000100 0000008010 00	Table 23
B	1b0000000100		Start scan 5.6
B	000000000000		Readiness check 5.6, repeated 4 times until successful
S	d70000006700		Read gain / offset values 5.8.2
E	dc0000001d00	7b1e2f168c 1017141000 0000212121 070000790b 14000f0000 00000000	Set exposure settings Table 23
I	080000000400		Transfer 4 lines / 42728 bytes of the calibration image 5.10
B	000000000000		Readiness check 5.6, repeated twice until successful
I	08000000ac00		Transfer 172 lines / 1837304 bytes of the calibration image 5.10
S	d70000006700		Read gain / offset values 5.8.2

Transaction types: B - Basic, E - Extra parameter, S - Status, I - Image

Table 31: Transactions of the scan (continued)

Type	Main parameters (hex)	Extra parameters (hex)	Description
E	dc0000001d00	7b1e2f168c 1017141000 0000212121 070000790b 14000f0000 00000000	Set exposure settings Table 23
I	080000000400		Transfer 4 lines / 42728 bytes of the calibration image 5.10
B	000000000000		Readiness check 5.6, repeated twice until successful
S	18000014dc00		Read row mask 5.8.2
S	0f0000001200		Get size of the main image Table 26. Returned status is bc011f01bc01... This means the image has 287 * 3 (color channels) = 861 lines of 444 bytes each.
B	000000000000		Readiness check 5.6, repeated 3 times until successful
I	08000000d800		Transfer 216 lines / 95904 bytes of the main image 5.11. Repeated 3 times. In this case, they are transferred in chunks of 216 lines. It would also be fine to use the simpler scheme with 255 lines / transaction.
I	08000000d500		Transfer the rest, that is 213 lines / 94572 bytes of the main image.

Transaction types: B - Basic, E - Extra parameter, S - Status, I - Image

6 SANE

SANE (abbreviated for "Scanner Access Now Easy") is the standard software package for accessing scanners in Linux-like operating systems. SANE was created because previous scanner software packages were often specific to a single device / vendor. Its most notable feature is the separation between backend (device driver) and frontend (user scanning software).

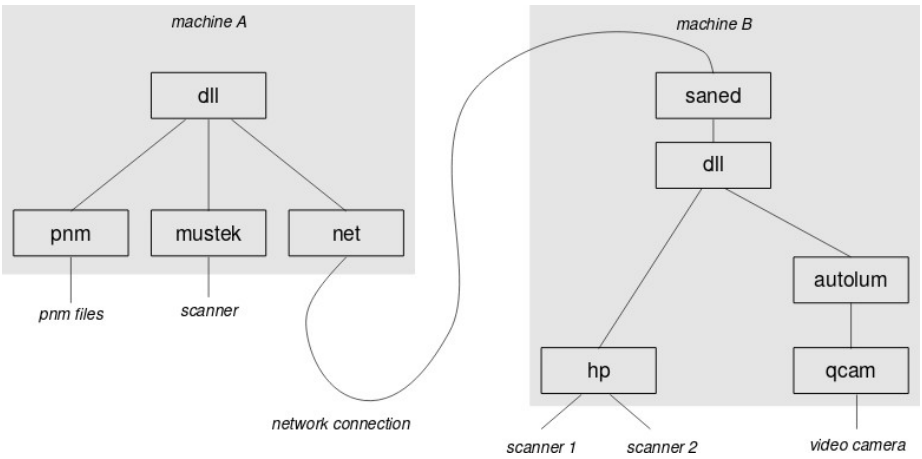
SANE contains many different backends for devices of different manufacturers. A lot of these drivers were not provided by manufacturers but instead

developed by hobbyists using various reverse-engineering procedures.

Another unique feature is *saned*, which is a server daemon that allows scanning over a network. This is desirable since one can share a scanner among multiple machines in the same location. When using *saned*, a special *net* backend is used on the client machine to make the remote scanner accessible just like a local one in the user's preferred frontend.

SANE frontends include *scanimage* for scanning on the command line, *xsane* for a graphical interface and a web application that can be served on a network and accessed via web browser, this is another way of providing shared scanner access.

Figure 23: Example SANE hierarchy



6.1 The SANE standard

As mentioned before, SANE consists of backends (that manage devices) and frontends (that provide applications). Any backend can be combined with any frontend thanks to the SANE interface / standard. In theory there could be multiple independent implementations of this standard, however only the official SANE package is commonly used.

The SANE interface is an application programming interface (API) consisting of specified C procedures / types that are implemented by backends and called by frontends. Frontend and backend can be combined either by static or dynamic linking. To make a frontend support multiple backends, the frontend can be linked to a meta-backend that finds out which backend is suitable for the currently attatched scanner, loads it via dynamic linking, and then mimics the behavior of the suitable backend.

[21, 3.1]

6.1.1 Primitive types

The standard defines various primitive data types (*SANE_Bool, SANE_Byte, SANE_Word, SANE_Fixed, SANE_String*) which have certain guaranteed qualities like minimum number of bytes, endianness etc. to migitate differences in C compilers and hardware.

There is also an enum type *SANE_Status* which operations use to report back on status.

6.1.2 Device descriptor

The device descriptor contains identifying information about a scanner (name / model), is returned by the *sane_get_devices* procedure and is used to obtain a device handle from the *sane_open* procedure.

6.1.3 Scanner handle

The scanner handle is a pointer that is returned by *sane_open* and is used to control the scanner using SANE operations (get / set options, start scan).

6.1.4 Options

Options are a powerful feature of the SANE standard in that the same type and procedures are used to control every setting on a device: basic ones like resolution or scan area, but also device-specific settings; this is independent of manufacturer / backend.

The option descriptor is a struct type with various components specifying an option's meaning and its possible values: name (unique identifier), title (human-readable name), description, type (bool / string etc.), unit (pixels / millimeters etc.) and constraints (f.e. a range of possible coordinate values in a dimension, a list of allowed string values).

[21, 4.2.9]

Every device handle has a certain number of option descriptors associated with it. An option descriptor can be fetched with the *sane_get_option_descriptor* procedure by specifying the index of the option. Backends can define any number of custom option descriptors, but some descriptors' meaning is specified in the standard:

Table 32: Special option descriptors specified in the SANE standard

Name	Description
Option count	Always available at index 0. Tells how many options there are in total.
resolution	Scan resolution (in dpi)
preview	Bool type, trade scan quality for speed when enabled
Scan area	Four options (*tl-x, tl-y, br-x, br-y*) that give the coordinates of a rectangle delimiting the scan area.

[21, 4.5]

Options values are queried/set using the *sane_control_option* procedure specifying the option index and a generic pointer to load / store the value. This also reports back on how well the request could be met (f.e. a resolution could not be set exactly) and whether the changed setting has changed the availablility of other options.

[21, 4.4]

6.1.5 Operations

The standard specifies various procedures for initialization, configuring options, scanning and retrievig data.

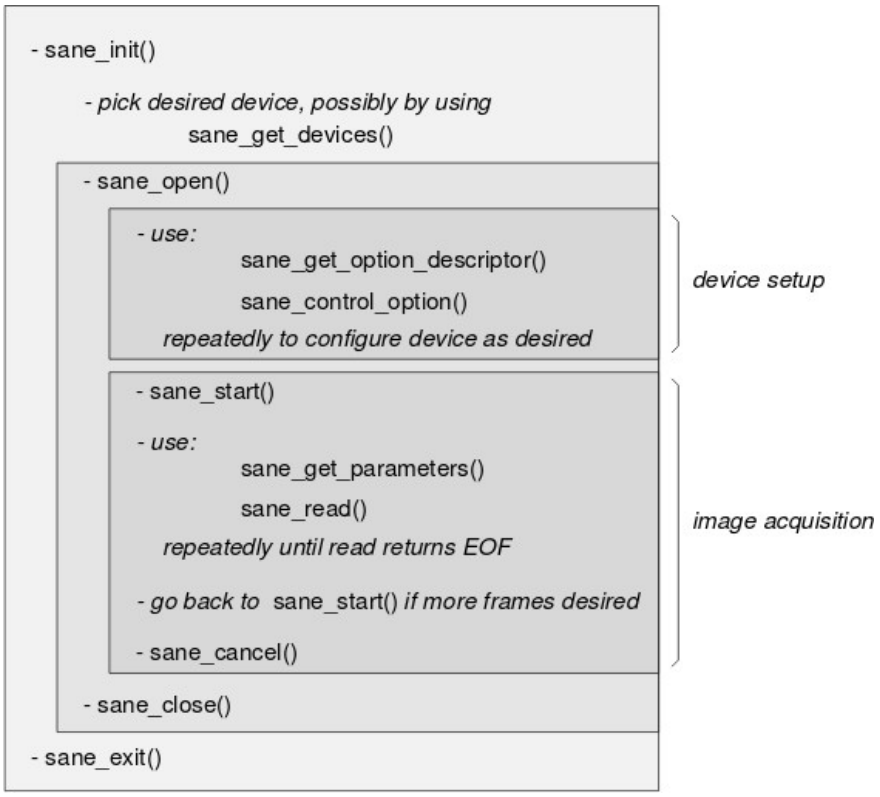
Table 33: Operations defined in the SANE standard (not exhaustive)

Name	Description
sane_init	Must be called before using other procedures of the backend
sane_get_devices	Get a list of device descriptors, one for each device the backend has found.
sane_open	Initialize the device with a specified descriptor. Returns
sane_close	Free scanner handle
sane_exit	Free data in backend
sane_get_option_descriptor	Get the option descriptor at the speficied index.
sane_control_option	Set or retrieve an option value at the specified option index.
sane_get_parameters	Get parameters (format / dimensions) of image returned by *sane_read*.
sane_start	Start scanning
sane_read	Read retrieved image bytes
sane_cancel	Cancel a scan in progress

[21, 4.3]

When using SANE, typical code flow might look as follows:

Figure 24: Typical code flow when using SANE



[21, 4.4]

6.1.6 Network protocol

This specifies how a SANE daemon should communicate with a client, possibly on a remote machine. This protocol is done in a RPC (remote-procedure-call) style: The client calls procedures on the server in a similar fashion to how they are called locally: The parameters are encoded in a packet, sent over the network, unpacked at the server, the procedure is executed there, the return value takes the reverse direction.

The exact encoding of packets is not specified (TCP/IP is not the only possible transport layer), for compatibility it seems reasonable to consider the official implementations of *saned* (server) and the *net* backend (client) .

[21, 5]

6.2 Developing a driver / backend

The SANE developers give some tips on writing a backend [22]:

- First, one should acquire information on how the device is controlled. This can be official documentation released by the manufacturer or obtained via various kinds of reverse engineering (software, firmware, sniffing, hardware...).
- It is advisable to first write a minimal, working standalone scanning program not part of SANE. This makes debugging easier.
- After the standalone program is working, one can start developing the backend: If the standalone program is already written in C, its code can be copied into the backend files and called by the public backend procedures.

[23] is focused on reverse-engineering Windows-only scanners for SANE. Despite being somewhat outdated and referring mostly to older SCSI scanners, the website provides useful information:

- As an alternative to sniffing, Windows drivers could be disassembled using special toolkits
- There is sniffing software running inside Windows (as an alternative to the Linux-based sniffing used here)
- Write down information about every scan run performed to keep track which settings give which recording
- Change only one option per scan run so the changes in traffic can be correlated with that option
- Keep scan area small to reduce recording size
- Constant bursts of traffic while idling may indicate polling for a pressed "start scan" button on the device
- Modern USB scanners have two levels of communication: One is how to set parameter strings and another is the meaning of the bytes inside the parameter string. Try to understand the first level first.
- If large chunks of data are transmitted, think about their meaning: What the driver sends could be firmware code, calibration data, motor acceleration data, gamma tables. The device could send calibration data or actual image data.

6.3 SANE and the CrystalScan 7200

6.3.1 pieusb

pieusb is a SANE backend that supports the CrystalScan 7200 among other scanners sold under the Reflecta brand. Development began in 2012 by Klaus Kaempf et al., in 2015 it got merged into mainline SANE code. Supported devices include:

- Reflecta CrystalScan 7200
- Reflecta ProScan 7200
- Reflecta 6000 Multiple Slide Scanner

The version of SANE / *pieusb* used here is **1.0.25-4.1**, as is part of Debian 9 (Stretch).

To use the *pieusb* driver, it must be enabled when compiling SANE (environment variable *BACKENDS*) and enabled in the dynamic loader backend 6 (*dll.conf*).

6.3.2 Fixing I/O errors

For my device, scanning unfortunately did not work as-is. When testing operation using *scanimage*, an "Error during device I/O" was reported. To find the cause of such problems, SANE can provide useful debugging output. This is controlled using environment variables to set a debugging verbosity level for various SANE subsystems. In this case, *SANE_DEBUG_PIEUSB=255* was set before running *scanimage* to get all possible messages from the driver.

```
[pieusb] sanei_pieusb_command() finished with state 4
[pieusb] sanei_pieusb_cmd_17 failed: 0x04
[pieusb] sane_start(): sanei_pieusb_cmd_17 failed: 4
scanimage: sane_start: Error during device I/O
```

Figure 25: Log of scanimage (unpatched SANE)

This shows the error happened in *sane_start*, while calling *sanei_pieusb_cmd_17*. This is only done once in *sane_start*:

```
998   if (status.pieusb_status != PIEUSB_STATUS_GOOD) {
999     DBG(DBG_error, "sane_start(): sanei_pieusb_cmd_17
          failed: %d\n", status.pieusb_status);
1000    return SANE_STATUS_IO_ERROR;
1001  }
```

Figure 26: First point of failure in pieusb.c

sanei_pieusb_cmd_17 does not transfer any scanning parameters, but it is performed presumably because the vendor software also performs it. The lines were commented out, SANE recompiled and *scanimage* run again:

```
[pieusb]       sanei_pieusb_command() finished with state 4
[pieusb] sane_start(): sanei_pieusb_cmd_slide failed: 4
scanimage: sane_start: Error during device I/O
```

Figure 27: Log of scanimage (after first patch)

sanei_pieusb_cmd_slide is called to turn on some lamp on the device:

```
1046    sanei_pieusb_cmd_slide (scanner->device_number,
            SLIDE_LAMP_ON, &status);
1047    if (status.pieusb_status != PIEUSB_STATUS_GOOD) {
1048      DBG (DBG_error, "sane_start(): "
              sanei_pieusb_cmd_slide failed: %d\n", status.
              pieusb_status);
1049      return SANE_STATUS_IO_ERROR;
1050    }
```

Figure 28: Second point of failure in pieusb.c

This section of code was commented out as well. After this patch, the scan completed successfully. No problems were observed when scanning with various parameters, so it seems the offending commands were superfluous for my device. It could be that there are multiple iterations of the CrystalScan 7200 that differ slightly in their I/O syntax.

6.3.3 Supported features

pieusb seems to support most if not all the features of this scanner. The following were tested and confirmed working:

- Scan resolution: 300 - 7200 dpi
- Color mode: Grayscale, RGB, RGB + infrared
- Bit depth (8 / 16 bit)
- Limiting scan area
- Exposure time, digitizer gain per channel
- Basic image processing: brightness / contrast, gamma, color balance
- Column brightness calibration
- Automatic IR dust correction

One downside is that the IR image can not be obtained easily as an extra channel / file for manual dust removal (or other uses of IR scanning). One has to rely on SANE's internal IR processing routines. It is possible to scan in RGB + infrared mode and apply no processing, but there seems to be a problem with output as the file is not larger than when scanning without IR.

7 Image processing

7.1 Calibration

7.2 Point operations

Point operations are a special class of image processing operations where the output value at any pixel only depends on the input value at that pixel (so the operation can be described as a function from the channel levels to the channel levels) [25]. For all described operations, if the output value lies outside $[0, 1]$, it is clipped to the respective limit.

7.2.1 Brightness

Adjusting brightness means adding an absolute number to each pixel value. To increase the brightness by k, a function $f(b) = b + k$ is used.

Figure 29: Brightness operation ($k = 0.3$)

7.2.2 Contrast

Changing contrast means compressing the brightness function so that its climb is steeper at the middle. To increase the contrast by a factor $k > 0$, a function $f(b) = (b - 0.5) * k + 0.5$ is used. The brightness function gets centered around zero, steepened by factor k, then moved back.

Figure 30: Contrast operation ($k = 2$)

7.2.3 Gamma

Changing gamma means bending the brightness function so more of its climb is in the dark / light area. To apply gamma correction for a $k > 0$, a function $f(b) = b^k$ is used. If $k < 1$, the function will be bent to the right so the majority of the climb happens in darker areas. If $k > 1$, the function will be bent to the right so the majority of the climb happens in darker areas.

Different image recording (film, digital sensors) and image display (screen, print) technologies have a different inherent gamma function (their input-output response curve) and gamma correction is neccessary to make images look good on different media.

Figure 31: Gamma operation ($k = 2$)

7.2.4 Levels

7.2.5 Color balance

7.2.6 Negative film

7.3 Infrared denoise

7.4 Automatic cropping

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

The device

[3] https://reflecta.de/en/products/detail/~id.15/reflecta-CrystalScan-7200.html

[4] http://www.filmscanner.info/en/ReflectaCrystalScan7200.html

[5] https://www.hamrick.com/

[6] http://www.silverfast.com/scanner-software/en.html

USB standard

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

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

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

USB sniffing

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

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

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

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

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

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

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

Vendor-supplied scanner software [language=Python, firstline=37, lastline=45]

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

Image editing

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

Developing USB drivers

[19] https://github.com/pyusb/pyusb
Control USB devices from Python. Has a great tutorial on using the software.

SANE

[20] http://www.sane-project.org/

[21] http://www.sane-project.org/sane.pdf
The SANE standard

[22] http://sane-project.org/backend-writing.txt
General guidelines on writing SANE backends

[23] http://www.meier-geinitz.de/sane/misc/develop.html
How to develop SANE backends without documentation (using reverse engineering)

[24] https://gitlab.com/sane-project/backends/tree/master/backend
Code repository (Git)

Image processing

[25] https://pippin.gimp.org/image-processing/chap_point.html