



**MATEMATICKO-FYZIKÁLNÍ
FAKULTA**
Univerzita Karlova

BAKALÁŘSKÁ PRÁCE

Peter Lakatoš

Analyzátor USB paketů

Katedra distribuovaných a spolehlivých systémů

Vedoucí bakalářské práce: Mgr. Pavel Ježek, Ph.D.

Studijní program: Informatika

Studijní obor: Programování a softwarové systémy

Praha 2021

Prohlašuji, že jsem tuto bakalářskou práci vypracoval(a) samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů. Tato práce nebyla využita k získání jiného nebo stejného titulu.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona v platném znění, zejména skutečnost, že Univerzita Karlova má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle §60 odst. 1 autorského zákona.

V dne

Podpis autora

Poděkování.

Název práce: Analyzátor USB paketů

Autor: Peter Lakatoš

Katedra: Katedra distribuovaných a spolehlivých systémů

Vedoucí bakalářské práce: Mgr. Pavel Ježek, Ph.D., Katedra distribuovaných a spolehlivých systémů

Abstrakt: USB zbernica je dnes jedným z najrozšírenejších spôsobov pripojenia periférií k počítaču. Cieľom práce bolo vytvoriť software, ktorý analyzuje zachytenú komunikáciu medzi zariadením pripojeným na danú zbernicu a počítačom.

Aplikácia následne rozumným spôsobom vizuálne zobrazuje zanalyzované dáta. Počiatočná verzia sa špecificky zameriava na HID triedu zariadení a ponúka aj sémantický význam jej úzkej podmnožiny do ktorej patria myš, klávesnica a joystick. Pri vizuálnej reprezentácii dát sa práca inšpiruje rôznymi dostupnými softwarmi, pričom rozlične kombinuje resp. dopĺňa ich vlastnosti a implementuje z nich tie, ktoré vníma ako najlepšie riešenie v danej situácii.

Ďalšie vlastnosti aplikácie sú napríklad parsovanie HID Report Descriptoru vďaka ktorému je jednoduchšie pridať sémantickú analýzu rôznym ďalším HID zariadeniam. Celkový návrh aplikácie by mal ponúknuť možnosť budúcej implementácie ďalších USB tried pre prípadné rozšírenie.

Klíčová slova: USB HID

Title: USB Packet Analyzer

Author: Peter Lakatoš

Department: Department of Distributed and Dependable Systems

Supervisor: Mgr. Pavel Ježek, Ph.D., Department of Distributed and Dependable Systems

Abstract: Abstract.

Keywords: key words

Obsah

1	Úvod	3
1.1	Základné pojmy	3
1.2	Existujúce aplikácie	8
1.3	Požadované funkcie	13
1.4	Ciele práce	15
2	USB a Windows	16
2.1	USB zbernica	16
2.2	Device object a device stack	16
2.2.1	Drivery	16
2.3	Komunikacia s USB zariadením	16
2.4	USB descriptor	16
2.4.1	Rozloženie USB zariadenia z hladiska descriptorov	16
2.5	HID zariadenia	17
2.5.1	Reporty	17
2.5.2	Report Descriptor	17
3	Analýza	18
3.1	Získanie USB packetov	18
3.2	Sémantická analýza dát	20
3.3	Voľba frameworku	24
3.4	Spracovávanie pcap súborov	25
3.5	Uchovávanie informácií	26
3.6	Zobrazenie základných informácií	27
3.7	Hexdump	30
3.8	Zobrazenie sémantického významu dát	32
4	Vývojová dokumentácia	34
4.1	Architektúra aplikácie	34
4.2	Jadro aplikácie	34
4.2.1	USB_Packet_Analyzer	34
4.2.2	Item Manager	34
4.2.3	DataViewer	34
4.2.4	TreeItem	34
4.3	Modely	34
4.3.1	AdditionaldataModel	34
4.3.2	ColorMapModel	34
4.3.3	DataViewerModel	34
4.3.4	TreeItemBaseModel	34
4.3.5	USBPcapHeaderModel	34
4.4	Interpretery	35
4.4.1	BaseInterpreter	35
4.4.2	Interpreter factory	35
4.4.3	Interpretery descriptorov	35
4.4.4	Interrupt transfer interpretery	35

4.5	Delegáti	35
4.6	HID	35
4.6.1	HIDDevices	35
4.7	Práca so súbormi	35
4.7.1	FileReader	35
4.8	Globálne dáta	36
4.8.1	ConstDataHolder	36
4.8.2	PacketExternStructs	36
5	Možnosti rozšírenia	37
5.1	Ukladanie výstupu do súboru	37
5.2	Iná vizuálna reprezentácia dát	37
5.3	Pridávanie nových interpreterov pre descriptor	37
5.4	Pridanie interpreteru na interrupt transfer	37
5.4.1	Pridanie nových HID zariadení	37
5.5	Pridanie analýzy pre isochronous a bulk transfer	37
5.6	?Možnosť rozšírenia na iné platformy?	37
6	Užívateľská dokumentácia	38
6.1	Inštalácia	38
6.2	Orientácia v GUI aplikácie	38
6.3	Používanie aplikácie	38
7	Záver	39
7.1	Zhrnutie	39
7.2	Budúce plány	39
	Zoznam použitej literatúry	40
	Zoznam obrázkov	44
	Zoznam tabuliek	46
	Seznam použitých zkratok	47
	Prílohy	48
.1	První příloha	49

1. Úvod

USB je najrozšírenejšia zbernica na pripojenie rôznych periférií k počítaču. Vznikla v druhej polovici 90. rokov 20. storočia, kedy boli rôzne zariadenia a ich porty veľmi úzko mapované. Na pripojenie základných zariadení ako myš alebo klávesnica slúžil napríklad sériový port PS/2 [1]. K pripojeniu tlačiarne sa často používal paralelný port Centronics [2]. Ešte pred PS/2 portom sa myš pripájala cez veľmi známy sériový port RS-232 [3]. Všetky tieto porty boli typu *point-to-point* – na jeden port je možné pripojiť len jedno zariadenie. Toto sa paralelným portom dalo čiastočne obísť tým, že niektoré zariadenia podporovali tzv. *daisy chain* – do pripojeného zariadenia sa pripojí ďalšie zariadenie, do toho sa pripojí ďalšie, atď. (napríklad typická tlačiareň toto nepodporovala, takže musela byť pripojená na konci *daisy chainu*). Existovala takisto paralelná *SCSI* zbernica [4], ktorej návrh bol prispôbený aby podporoval *daisy chain*. Tá fungovala dobre na zariadeniach ako externé HDD a skenery, ale bola nepraktická pre zariadenia ako myš a klávesnica.

USB vzniklo za účelom nahradiť a zjednotiť tieto spôsoby pripojenia bežných periférií k počítaču. Návrh zbernice je založený na hviezdicovej topológii, ktorá umožňuje cez jeden port pripojiť až 127 zariadení súčasne. Z toho vyplýva, že USB interface v sebe zahŕňa obrovskú množinu protokolov, ktoré sú hierarchicky usporiadané. K analýze paketov ktoré sa pohybujú na danej zbernici nám slúžia tzv. USB paket analyzátory. Tie môžu mať podobu hardwarového zariadenia, alebo softwarovej aplikácie. Môžu slúžiť napríklad ako učebná pomôcka pre účel lepšieho pochopenia jednotlivých protokolov. Takisto sa často využívajú pri implementácii vlastného USB zariadenia na ladenie komunikácie medzi daným zariadením a driverom. Využitie ale majú aj v opačnom prípade, keď implementujeme vlastný driver a potrebujeme sledovať jeho komunikáciu s konkrétnym zariadením. Cieľom tejto práce bude naprogramovať funkčný softwarový USB analyzátor, ktorý by mal presnejšie slúžiť práve ako učebná pomôcka pre lepšie pochopenie určitej množiny protokolov. Cieľová platforma aplikácie bude Windows.

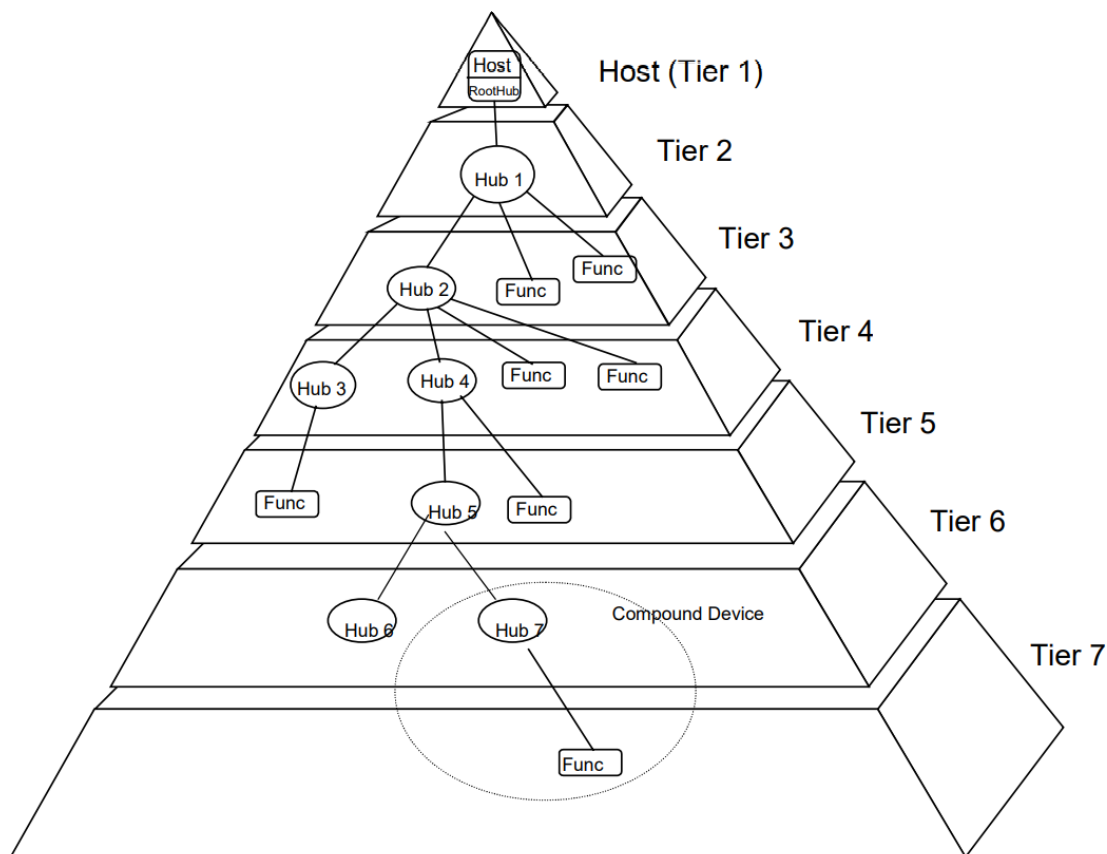
1.1 Základné pojmy

V tejto sekcii si vysvetlíme niektoré základné pojmy ktoré budeme neskôr v texte používať.

Ako sme už vyššie spomínali a ako ilustruje obrázok 1.1, USB zbernica je založená na vrstevnatej hviezdicovej topológii. Na vrchu všetkého sa nachádza **USB Host** [6] čo je systém do ktorého sa pripájajú ostatné USB zariadenia (v našom prípade je *USB Host* počítač). **USB zariadenie** [7] je buď:

- **Hub** [8] – poskytuje dodatočné pripojenia k USB zbernici.
- **Funkcia** [9] – poskytuje novú funkcionálnu systém (napríklad joystick, reproduktory, myš a pod.)

V každom USB systéme sa nachádza práve jeden *USB Host*. Ten má integrovaný tzv. **Root Hub** [6], ktorý poskytuje možné body pripojenia pre ďalšie zariadenia. Interface medzi hostom a USB sa nazýva **Host Controller** [6]. Vzhľadom



Obr. 1.1: USB topológia. Obrázok prevzatý z USB 2.0 špecifikácie [5].

na niektoré časové obmedzenia USB je maximálny počet vrstiev 7 (vrátane *USB Host* vrstvy). Každý káblový segment je *point-to-point* [10] spojenie medzi:

- $host \longleftrightarrow hub/funkcia$
- $hub \longleftrightarrow hub/funkcia$

USB zariadenia využívajú tzv. **descriptor** na predávanie informácií o sebe samých. **Descriptor** [11] je dátová štruktúra s predom definovaným formátom. Existuje viacero typov *USB descriptorov* (device, endpoint, interface atď.), ktorých význam si vysvetlíme neskôr.

Vzhľadom na hierarchickú štruktúru USB protokolov sa *USB zariadenia* delia na rôzne triedy [12]. USB trieda je zoskupenie zariadení (alebo interfacov) ktoré majú spoločné vlastnosti alebo funkcionality. Tieto triedy umožňujú USB hostovi identifikovať dané zariadenie a jeho funkcionality. Každá trieda má svoju vlastnú *Class Specification* – definuje správanie zariadení v jednotlivých triedach a opisuje ich komunikačný protokol, ktorý sa naprieč triedami líši. Takisto definuje rôzne descriptors, ktoré sú špecifické pre danú triedu. Príklady USB tried a jednotlivých zariadení ktoré do nich patria sú:

- Mass Storage (napr. SD karta a flash disk)
- Audio (napr. reproduktory a slúchadlá)
- HID – Human Interface Device (napr. myš, klávesnica alebo joystick)

Paket [13] je súbor dát zoskupený na prenos po zbernici. Typicky sa skladá z troch častí:

- základné informácie o danom pakete (napríklad zdroj, cieľ, dĺžka) – takisto nazývané hlavička paketu
- samotné dáta
- detekcia chýb, opravné bity

Komunikácia na zbernici medzi *USB hostom* a *zariadením* prebieha práve pomocou prenosu *USB paketov*. Existujú 4 typy takýchto prenosov [14]:

- **Control Transfer** – používa sa na nakonfigurovanie USB zariadenia v momente keď sa pripojí na zbernicu.
- **Bulk Data Transfer** – typicky pozostáva z väčšieho množstva dát ktoré sú posielané nárazovo (využívajú ho najmä tlačiarne alebo skener). Vďaka detekcii chýb na hardwarovej úrovni je zaistená správnosť prenesených dát.
- **Interrupt Data Transfer** – spoľahlivý prenos ktorý sa využíva hlavne na odovzdávanie aktuálnych informácií (ako napríklad pohyb myšou). Tieto informácie musia byť doručené USB zbernicou za čas kratší ako má špecifikované dané zariadenie.
- **Isochronous Data Transfer** – takisto nazývaný ako streaming v reálnom čase. Typický príklad je prenos zvuku.

Našu aplikáciu by sme chceli zamerať na Windows a tak si vysvetlíme ešte zopár špecifických pojmov, ktoré sa viažu na túto konkrétnu platformu.

Podľa MSDN dokumentácie [15] je **USB client driver** software nainštalovaný na počítači, ktorý komunikuje s USB zariadením aby spojzdnil jeho funkcionality. Žiaden *USB client driver* ale nemôže priamo komunikovať so svojím zariadením. Namiesto toho vytvorí požiadavku, súčasťou ktorej je dátová štruktúra nazývaná **URB** [16] (USB Request Block). Tá opisuje detaily požiadavky, takisto ako aj status o jeho vykonaní.

Na záver si ešte zdefinujeme rozdiel medzi *USB paket analyzátorom* a *USB paket snifferom*. Pod pojmom **USB paket sniffer** budeme rozumieť aplikáciu, ktorá monitoruje dianie na USB zbernici a je schopná ho rozumným spôsobom ukladať v predom definovanom formáte. Ako **USB paket analyzátor** budeme brať aplikáciu ktorá je schopná rozanalyzovať USB pakety (istým spôsobom ich vyobraziť alebo ukázať ich sémantický význam) uložené v predom definovanom formáte. Bežne sa tieto pojmy označujú za jednu a tú istú vec, aj keď ich funkcionality spolu nijako priamočiaro nesúvisí a existujú nástroje, ktoré vedia len jedno alebo druhé. Preto dáva zmysel ich od seba explicitne oddeliť.

Momentálne by sme mali chápať všetky základné pojmy týkajúce sa USB, a tak si poďme trochu bližšie objasniť zameranie našej aplikácie. Našu aplikáciu zameriavame výukovým smerom pre programátorov, ktorí chcú lepšie pochopiť komunikáciu na USB zbernici. Z toho dôvodu by sme v nej určite chceli zahrnúť analýzu základných USB descriptorov, ktoré sú bližšie definované v špecifikácii USB 2.0 [17] v kapitole 9.6. Keďže chceme bližšie priblížiť komunikáciu na danej zbernici, potrebujeme konkrétne zariadenia, s ktorými ju budeme analyzovať.

Dáva dobrý zmysel si zvoliť zariadenia, ktoré každý z nás dobre pozná, má ich k dispozícii a bežne ich využíva. Zároveň by ale mali mať dostatočne jednoduchý komunikačný protokol. Práve preto sa s našou aplikáciou zameriame na užšiu podmnožinu HID zariadení, konkrétne myš, klávesnica a joystick. Vzhľadom na zameranie našej aplikácie výukovým smerom prikladáme najväčšiu prioritu samotnej analýze dát. Z dôvodu celkovej univerzality USB je z didaktického hľadiska ťažké nasimulovať jednotný príklad u každého študenta zvlášť. Už len obyčajná myš, aj keď je to jedno zariadenie, má od rôznych výrobcov inak nadefinované správanie a posiela dáta v rozličných formátoch. Preto je pre nás dôležité vedieť analyzovať pakety, ktoré si učiteľ predpripraví, skontroluje ich didaktickú správnosť a uloží do súboru. Podpora živého zachytávania paketov a ich analýzy je tak v našom programe najmenej dôležitá.

Keďže sa v našej práci budeme venovať hlavne analýze HID zariadení, tak si túto USB triedu rozoberieme trochu detailnejšie.

HID

Podľa dodatku k USB špecifikácii [18] je **HID** (z anglického „Human Interface Device“) USB trieda pozostávajúca prevažne zo zariadení, ktoré sú využívané človekom na riadenie určitých systémových aplikácií. Medzi najpoužívanjšie príklady patrí myš, klávesnica alebo joystick.

Ako sme už spomínali vyššie, jednotlivé USB triedy majú definované vlastné descriptorý špecifické pre danú triedu. Jedným z takýchto descriptorov je aj *Report Descriptor*. Ten popisuje dáta, ktoré generuje konkrétne zariadenie. Analýzou *Report Descriptoru* sme schopní určiť veľkosť a kompozíciu dát posielaných zariadením. Z toho vyplýva, že komunikácia HID zariadením s USB hostom sa môže líšiť nie len vo veľkosti posielaných dát, ale takisto aj v ich význame.

Lepšie to uvidíme na konkrétnom príklade. K dispozícii máme 2 rozdielne myši – Genius DX-120 [19] a Logitech G502 Proteus Spectrum [20]



(a) Fotka genius myši prevzatá z oficiálnej (b) Fotka logitech myši prevzatá zo stránky
genius stránky [21] obchodu [22]

Obr. 1.2: Ukážka myší, ktorých input budeme porovnávať

Teraz si ukážeme ako sa líši ich input. Dáta budeme vyobrazovať pomocou obyčajného hexdumpu (zvýraznená časť v hexdumpe reprezentuje input zariadenia). Na oboch myšiach stlačíme ľavé tlačidlo a mierne ich posunieme smerom

hore. Dáta, ktoré poslala genius myš sú vyobrazené na obrázku 1.3 a dáta poslané logitech myšou môžeme vidieť na obrázku 1.4.

```
0010 01 01 00 02 00 81 01 04 00 00 00 01 00 ff 00 ..... .....
```

Obr. 1.3: Ukážka hexdumpu so zvýrazneným inputom genius myši.

```
0010 01 01 00 03 00 81 01 08 00 00 00 01 00 00 00 ff ..... .....
```

```
0020 ff 00 00 ...
```

Obr. 1.4: Ukážka hexdumpu so zvýrazneným inputom logitech myši.

Aj napriek tomu, že sa jedná o zariadenia z tej istej USB triedy a dokonca o rovnaké zariadenie – myš, má ich komunikácia rozličný tvar definovaný priamo výrobcom zariadenia. Analýzou *Report Descriptor* (o ktorej si viac povieme neskôr v práci) sme zistili, že dáta, ktoré poslala genius myš majú nasledujúci význam:

- Byte 0: bity 0–2 reprezentujú stlačenie jednotlivých tlačidiel, bity 3–7 tvoria len dodatočnú výplň bytu
- Byte 1: reprezentuje súradnicu X
- Byte 2: reprezentuje súradnicu Y
- Byte 3: reprezentuje koliesko myši

Pre porovnanie, význam dát poslaných logitech myšou je nasledovný:

- Byte 0–1: reprezentujú stlačenie jednotlivých tlačidiel
- Byte 2–3: reprezentuje súradnicu X
- Byte 4–5: reprezentuje súradnicu Y
- Byte 6: reprezentuje koliesko myši
- Byte 7: je rezervovaný výrobcom myši

Vizuálne zobrazený význam dát genius myši je ukázaný na obrázku 1.5 a logitech myši na obrázku 1.6.

```
0010 01 01 00 02 00 81 01 04 00 00 00 01 00 ff 00 ..... .....
```

Tlačidlá X Y Koliesko

Obr. 1.5: Ukážka hexdumpu so zvýrazneným inputom genius myši s významom.

```
0010 01 01 00 03 00 81 01 08 00 00 00 01 00 00 00 ff ..... .....
```

```
0020 ff 00 00
```

Tlačidlá X Y Koliesko Reserved

Obr. 1.6: Ukážka hexdumpu so zvýrazneným inputom logitech myši s významom.

Z toho vyplýva, že aby sme boli schopní vykonať sémantickú analýzu HID zariadení, bude pre nás kľúčové vedieť rozparsovať *Report Descriptor* a na základe toho interpretovať input zariadení.

1.2 Existujúce aplikácie

Momentálne existuje niekoľko známych aplikácií ktoré slúžia na analýzu USB paketov. Ich predbežným skúmaním a používaním sme ale zistili, že úplne nevyhovujú našim konkrétnym požiadavkám. Avšak mnohé ich funkcie nám prídu užitočné a môžu poslúžiť ako inšpirácia v implementovaní našej aplikácie. V tejto kapitole si ukážeme výhody a nevýhody zopár aplikácií, ktoré sme si zvolili ako príklady v oblasti paket analyzátorov. Ich výber spočíval v tom, že sú veľmi rozšírené medzi verejnosťou a sú najbližšie k tomu čo by sme chceli od našej aplikácie.

Je nutné upozorniť, že väčšina dnešných analyzátorov sú platené aplikácie, prípadne majú odomknuté len základné vlastnosti s možnosťou dokúpenia si plnej verzie. Práve preto sme nemali možnosť si pri všetkých vyskúšať ich celú funkcionality a na niektoré platené funkcie máme tak len ilustračný pohľad.

Wireshark

Aplikácia, ktorá na prvý pohľad nesúvisí s USB zbernicou. Wireshark je pravdepodobne najznámejší analyzátor a sniffer sieťových paketov. Jeho funkcionality je veľmi rozsiahla, a vzhľadom na to, že sa jedná o open-source projekt, neustále rastie. Vďaka jeho obecnému návrhu podporuje spoluprácu s rôznymi inými sniffermi (LANalyzer, NetXRay a pod.). Jeden z takýchto snifferov je *USBPcap*, ktorý zachytáva USB komunikáciu a tým pádom je Wireshark schopný analyzovať pakety aj nad touto zbernicou.

Pre priblíženie niektorých funkcií Wiresharku si ukážeme analýzu komunikácie s USB myšou (Genius DX-120 [19]). Medzi tie úplne základné funkcie určite patrí hexdump dát nad ktorými prebieha analýza, ktorý je vyobrazený na obrázku 1.7.

```
0000 1c 00 a0 49 1f 6a 8a dc ff ff 00 00 00 00 08 00 ...I.j...
0010 01 01 00 06 00 80 02 34 00 00 00 03 09 02 34 00 .....4.....4
0020 02 01 00 a0 32 09 04 00 00 01 03 01 02 00 09 21 ....2.....!
0030 11 01 00 01 22 38 00 07 05 81 03 04 00 0a 09 04 ...."8.....
0040 01 00 00 03 00 00 00 09 21 11 01 00 01 22 16 00 .....!".....
```

Obr. 1.7: Ukážka hexdumpu vo Wiresharku.

Tento hexdump je tvorený dátami z jedného control prenosu, kde zariadenie posiela informácie o sebe samom v podobe rôznych descriptorov.

V hexdumpe si takisto vieme pomocou kliknutia a ťahania myšou označiť ľubovoľné dáta, ktoré chceme. Zvýraznené byty na obrázku 1.8 reprezentujú jeden *endpoint descriptor*.

```
0000 1c 00 a0 49 1f 6a 8a dc ff ff 00 00 00 00 08 00 ...I.j...
0010 01 01 00 06 00 80 02 34 00 00 00 03 09 02 34 00 .....4.....4
0020 02 01 00 a0 32 09 04 00 00 01 03 01 02 00 09 21 ....2.....!
0030 11 01 00 01 22 38 00 07 05 81 03 04 00 0a 09 04 ...."8.....
0040 01 00 00 03 00 00 00 09 21 11 01 00 01 22 16 00 .....!".....
```

Obr. 1.8: Ukážka hexdumpu so zvýrazneným endpoint descriptorom.

Pri pohybe myšou nad daným hexdumpom ponúka Wireshark interaktívnu odozvu, pričom farebne oddeľuje jednotlivé byty podľa ich významu. Na obrázku 1.9 vidíme konkrétny príklad – ak podržíme myš nad hexa častou bytu

00, automaticky nám to označí aj byte 04 pred ním, pretože spoločne reprezentujú jednotnú informáciu – položku *wMaxPacketSize* v *endpoint descriptor*.

```

0000  1c 00 a0 49 1f 6a 8a dc ff ff 00 00 00 00 08 00  ...I-j-...
0010  01 01 00 06 00 80 02 34 00 00 00 03 09 02 34 00  .....4....4-
0020  02 01 00 a0 32 09 04 00 00 01 03 01 02 00 09 21  ...-2-.....!
0030  11 01 00 01 22 38 00 07 05 81 03 04 00 0a 09 04  ..."8-...
0040  01 00 00 03 00 00 00 09 21 11 01 00 01 22 16 00  .....!...."

```

Obr. 1.9: Ukážka hexdumpe s farebným oddelením na základe významu.

Ďalšia užitočná vlastnosť je, že pri označení hexa znakov v hexdumpe, sa samé označia aj im odpovedajúce tlačiteľné znaky (obdobne to funguje aj opačným smerom). To, že vyššie označených 7 bytov na obrázku 1.8 reprezentujú *endpoint descriptor* sme zistili vďaka špecifikácii jednotlivých descriptorov a vlastnou analýzou bytov v hexdumpe. Wireshark ale ponúka rozličné zobrazenie tých istých dát, a to napríklad aj pomocou stromovej štruktúry, ktorá už jednotlivým bytom pridáva ich sémantický význam v slovnom tvare ako je ukázané na obrázku 1.10 nižšie.

```

> Frame 6: 80 bytes on wire (640 bits), 80 bytes captured (640 bits)
> USB URB
> CONFIGURATION DESCRIPTOR
> INTERFACE DESCRIPTOR (0.0): class HID
> HID DESCRIPTOR
> ENDPOINT DESCRIPTOR
> INTERFACE DESCRIPTOR (1.0): class HID
> HID DESCRIPTOR

```

Obr. 1.10: Ukážka reprezentácie dát pomocou stromovej štruktúry.

Jednotlivé položky si môžeme bližšie rozbaľiť. Napríklad vyššie zvýraznených 7 bytov reprezentujú konkrétny *endpoint descriptor*, ktorý je ukázaný na obrázku 1.11. Na tom istom obrázku si takisto môžeme všimnúť, že položka *wMaxPacketSize* má hodnotu 4, čo je presne hodnota bytov 04 00, ktoré sme spomínali vyššie na obrázku 1.9

```

▼ ENDPOINT DESCRIPTOR
  bLength: 7
  bDescriptorType: 0x05 (ENDPOINT)
  > bEndpointAddress: 0x81 IN Endpoint:1
  > bmAttributes: 0x03
  > wMaxPacketSize: 4
  bInterval: 10

```

Obr. 1.11: Endpoint descriptor reprezentovaný dátami zvýraznenými na obrázku 1.7 vyššie.

Medzi viac špecifické funkcie patrí detailnejšie vyobrazenie jednotlivých bytov a ich význam, ako je možné vidieť nižšie na obrázku 1.12. Na tomto obrázku vidíme rozbalenú položku *bEndpointAddress*, ktorej hodnota je 0x81. Siedmy bit tejto hodnoty reprezentuje smer endpointu (IN – slúži na prenos dát device → host, OUT opačne) a dolné 4 bity označujú číslo endpointu. Túto vlastnosť aj napriek jej využitiu mnohé konkurenčné aplikácie postrádajú.

```

    ▾ bEndpointAddress: 0x81 IN Endpoint:1
      1... .... = Direction: IN Endpoint
      .... 0001 = Endpoint Number: 0x1

```

Obr. 1.12: Ukážka vyobrazenia jednotlivých bytov.

Wireshark ponúka interaktívne užívateľské rozhranie. V prípade kliknutia na konkrétny byte v hexdumpe sa nám označí jemu odpovedajúca položka v stromovej štruktúre. Príklad je ukázaný na obrázku 1.13.

```

    ▾ ENDPOINT DESCRIPTOR
      bLength: 7
      bDescriptorType: 0x05 (ENDPOINT)
      > bEndpointAddress: 0x81 IN Endpoint:1
      > bmAttributes: 0x03
      ▾ wMaxPacketSize: 4
        ...0 0... .... = Transactions per microframe: 1 (0)
        .... ..00 0000 0100 = Maximum Packet Size: 4
      bInterval: 10

```

00 00 01 03 01 02 00 09 21
 07 05 81 03 04 00 0a 09 04
 09 21 11 01 00 01 22 16 00

Obr. 1.13: Ukážka kliknutia na položku v hexdumpe.

Podobne to funguje aj opačne, takže ak klikneme na položku v stromovej štruktúre, označí sa jej odpovedajúca časť v hexdumpe. Príklad kliknutia na *endpoint descriptor* v stromovej štruktúre a označenia jemu odpovedajúcej časti hexumpu je vidieť na obrázku 1.14.

```

> ENDPOINT DESCRIPTOR (1.0): class HID
> INTERFACE DESCRIPTOR (1.0): class HID
> HID DESCRIPTOR

```

00 00 01 03 01 02 00 09 21-2-... ..!
 07 05 81 03 04 00 0a 09 04-8-... ..
 09 21 11 01 00 01 22 16 00!.....

Obr. 1.14: Ukážka kliknutia na položku *endpoint descriptoru* v stromovej štruktúre.

Obecné vyobrazenie pohybu paketov na zbernici bez hlbšej analýzy je ukázané na obrázku 1.15 nižšie.

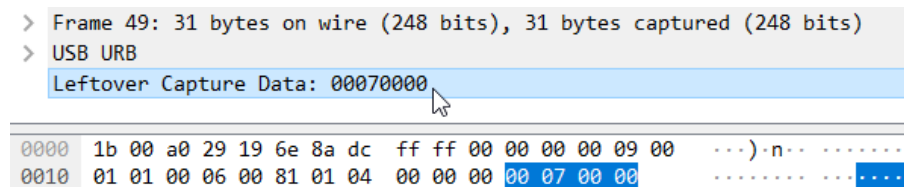
No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	host	1.6.0	USB	36	GET_DESCRIPTOR Request DEVICE
2	0.000684	1.6.0	host	USB	46	GET_DESCRIPTOR Response DEVICE
3	0.000750	host	1.6.0	USB	36	GET_DESCRIPTOR Request CONFIGURATION
4	0.001388	1.6.0	host	USB	37	GET_DESCRIPTOR Response CONFIGURATION
5	0.001432	host	1.6.0	USB	36	GET_DESCRIPTOR Request CONFIGURATION
6	0.002912	1.6.0	host	USB	80	GET_DESCRIPTOR Response CONFIGURATION
7	0.004170	host	1.6.0	USB	36	SET_CONFIGURATION Request
8	0.004827	1.6.0	host	USB	28	SET_CONFIGURATION Response
9	0.004905	host	1.6.0	USB	27	Unknown type 7f

Obr. 1.15: Príklad obecného vyobrazenia jednotlivých paketov vo Wiresharku.

Výhoda Wiresharku je hlavne v tom, že podporuje širokú škálu descriptorov a plná verzia programu je dostupná úplne zadarmo. Z pohľadu užívateľa je až prekvapivé, že aj napriek rozsiahlosti programu je aplikácia veľmi user-friendly orientovaná a dopĺňa ju intuitívne užívateľské rozhranie.

Naopak, jeho nevýhodou je sčasti neprehľadný hexdump. Ako môžeme vidieť na obrázku 1.7, jedná sa o obyčajný hexdump, ktorý nijakým spôsobom neoddeľuje význam dát bez interakcie užívateľa. Preto v momente ak by sme nemali

stromovú štruktúru k odpovedajúcemu hexdumpu, museli by sme sa riadiť špecifikáciou a vlastnou analýzou. V prípade rozsiahlejšieho hexdumpu môže byť veľmi obtiažné sa v ňom potom zorientovať. Ďalšia vec ktorá nám nevyhovuje, je chýbajúca sémantická analýza inputu rôznych zariadení. Ten je vyobrazený len pomocou hexdumpu a popisu „Leftover Capture Data“ ako je ukázané na obrázku 1.16. Zo sekcie 1.1 nám je teda jasné, že vôbec netušíme čo jednotlivé dáta znamenajú, pretože ich význam je definovaný v *Report Desriptore*.

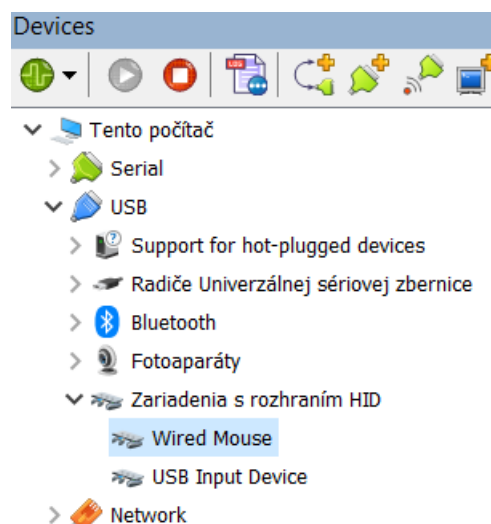


Obr. 1.16: Príklad inputu myši vo Wiresharku.

Device Monitoring Studio

Aplikácia ponúka analýzu sieťových a USB paketov, tak ako aj analýzu komunikácie prebiehajúcej cez sériový port. Zároveň slúži aj ako sniffer na všetkých týchto portoch.

Ako prvé na aplikácii zaujme spôsob zvolenia si zariadenia s ktorým bude sledovaná komunikácia. Je implementovaný štýlom stromovej štruktúry ako je ukázané na obrázku 1.17 nižšie, kde máme konkrétne označenú rovnakú myš s ktorou komunikáciu sme sledovali predchádzajúcim programom.



Obr. 1.17: Ukážka stromovej štruktúry na zvolenie si zariadenia, s ktorým bude zachytávaná komunikácia.

Základná verzia programu ponúka vizuálne zobrazenie *URB*, tak ako aj analýzu jednotlivých paketov. Pod analýzou si tu môžeme predstaviť ale len obyčajný hexdump, ktorý neposkytuje žiadne významové oddelenie dát a tým pádom je obtiažnejšie sa v ňom zorientovať. Príklad môžeme vidieť na obrázku 1.18.


```

05 01 09 02 A1 01 09 01 A1 00 05 09 19 01 29 03 .....~...~.....) .
15 00 25 01 75 01 95 03 81 02 75 05 95 01 81 01 ..%.u.*. .u.*. .
05 01 09 30 09 31 15 81 25 7F 75 08 95 02 81 06 ...0.1. %u.*. .
09 38 95 01 81 06 C0 C0 .8*. .Ř

```

Obr. 1.18: Príklad hexdumpu v Device Monitoring Studio.

Takisto tu nemáme kompletne sémantické vysvetlenie čo dané dáta znamenajú (napríklad pomocou stromovej štruktúry ako to rieši konkurencia). K dispozícii máme len veľmi obmedzený popis jednotlivých paketov (číslo paketu, device request, a pod.), pričom ani nie je veľmi jasné odkiaľ sa tieto informácie vzali. Príklad takéhoto popisu aj s hexdumpom je ukázaný na obrázku 1.19 nižšie.

```

000081: Get Descriptor Request (UP), 2021-04-13 10:17:37,4763790 +0,0000026. (1. Device: Wired Mouse) Status: 0x00000000
Descriptor Type: Device
Descriptor Index: 0x0
Transfer Buffer Size: 0x12 bytes

12 01 10 01 00 00 00 08 58 04 86 01 58 24 04 28 .....X.+XS.(
00 01 ..

```

Obr. 1.19: Príklad analýzy paketov.

Vyobrazenie *URB* (obrázok 1.20) tak ponúka súhrn týchto popisov jednotlivých paketov, ktoré sú postupne zachytené počas komunikácie na zbernici.

```

000222: Control Transfer (UP), 2021-04-05 14:52:08,3172528 +0,0003705. (1. Device: ) Status: 0x00000000
Pipe Handle: Control Pipe
18 03 ..
Setup Packet

000223: Bulk or Interrupt Transfer (UP), 2021-04-05 14:52:10,2584548 +1,9412020. (1. Device: ) Status: 0x00000000
Pipe Handle: 0xbaed3cb0 (Endpoint Address: 0x81)
Get 0x4 bytes from the device

000224: Bulk or Interrupt Transfer (DOWN), 2021-04-05 14:52:10,2584690 +0,0000142 (1. Device: )
Pipe Handle: 0xbaed3cb0 (Endpoint Address: 0x81)
Get 0x4 bytes from the device

```

Obr. 1.20: Ukážka vyobrazenia URB.

Pričom pri dvojkliku na šedé časti textu (napríklad *Setup Packet* alebo *Endpoint Address*) sa užívateľovi rozbalí okno s detailnejším popisom.

Analýza inputu myši, ktorú môžeme vidieť na obrázku 1.21, je riešená podobným spôsobom ako pri analýze descriptorov.

```

00000135 | 2021-04-13 11:57:08,8011899 | +0,0078834 | UP | 0x00000000 | URB_FUNCTION_BULK_OR_INTERRUPT_TRANSFER
/ Complete \
000135: Bulk or Interrupt Transfer (UP), 2021-04-13 11:57:08,8011899 +0,0078834. (1. Device: Wired Mouse)
Pipe Handle: 0x19297890 (Endpoint Address: 0x81)
Get 0x4 bytes from the device
00 00 01 00 .....

```

Obr. 1.21: Príklad inputu myši v Device Monitoring Studio.

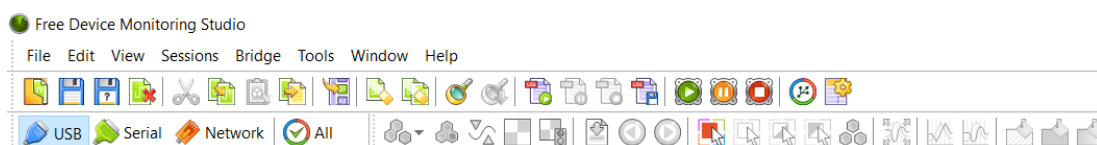
Obecné vyobrazenie jednotlivých paketov bez bližšej analýzy je riešené podobne ako vo Wiresharku, pričom pakety sú farebne oddelené podľa ich smeru pohybu na zbernici (posielané smerom host → zariadenie/smerom zariadenie → host). Toto je veľmi pekná funkcionálna, ktorá celkovo sprehľadňuje komunikáciu zariadenia s hostom. Príklad je ukázaný na obrázku 1.22.

00000073	2021-04-13 10:17:35.1931283	+25.9758328	UP	0xc0000011	URB_FUNCTION_BULK_OR_INTERRUPT_TRANSFER
00000074	2021-04-13 10:17:35.1932268	+0.0000985	UP	0xc0010000	URB_FUNCTION_BULK_OR_INTERRUPT_TRANSFER
00000075	2021-04-13 10:17:35.2130691	+0.0198423	UP		PnP: Device Surprise Removal
00000076	2021-04-13 10:17:35.2249868	+0.0119177	DOWN	0xffffd184	URB_FUNCTION_ABORT_PIPE
00000077	2021-04-13 10:17:35.2250067	+0.0000199	UP	0x80000300	URB_FUNCTION_ABORT_PIPE
00000078	2021-04-13 10:17:35.2250219	+0.0000152	UP		PnP: Device Disconnected
00000079	2021-04-13 10:17:37.4763688	+2.2513469	UP		PnP: Device Connected
00000080	2021-04-13 10:17:37.4763764	+0.0000076	DOWN	0x00000000	URB_FUNCTION_GET_DESCRIPTOR_FROM_DEVICE
00000081	2021-04-13 10:17:37.4763790	+0.0000026	UP	0x00000000	URB_FUNCTION_GET_DESCRIPTOR_FROM_DEVICE
00000082	2021-04-13 10:17:37.4763850	+0.0000060	DOWN	0x00000000	URB_FUNCTION_GET_DESCRIPTOR_FROM_DEVICE
00000083	2021-04-13 10:17:37.4763874	+0.0000024	UP	0x00000000	URB_FUNCTION_GET_DESCRIPTOR_FROM_DEVICE

Obr. 1.22: Príklad obecného vyobrazenia jednotlivých paketov v Device Monitoring Studio.

Zaujímavá funkcionálnosť, ktorú ale program ponúka len v platenej verzii, je umožnenie užívateľovi priamo komunikovať so zvoleným zariadením. Môžeme mu tak posilať rôzne požiadavky (niektoré z nich sú spomenuté v USB 2.0 špecifikácii[17] v kapitole 9.4) ako napríklad *GET_REPORT* kde špecifikujeme *Report ID* a prípadné ďalšie parametre, a zariadenie nám patrične odpovie.

Užívateľské rozhranie vyobrazené nižšie pomocou obrázku 1.23, pozostáva z pomerne veľa ikoniek a celkovo sa javí ako trochu neprehľadné. Pri prvotnej interakcii s programom chvíľu trvá, kým človek nájde čo i len základné informácie ako napríklad hlavičky ku jednotlivým paketom. Nepoteší ani fakt, že verzia zadarmo nedovoľuje monitorovanie dlhšie ako 10 minút a maximálny počet monitorovaní za jeden deň je taktiež 10.



Obr. 1.23: Užívateľské rozhranie Device Monitoring Studio.

1.3 Požadované funkcie

Ako prvé by sme si mali zadať platformu na ktorú budeme cieľiť s našou aplikáciou:

P1 Cieľová platforma našej aplikácie by mala byť Windows.

Keďže má naša aplikácia mať výukový charakter, tak sa pozrieme na typický výukový scénár jej používania. Učiteľ si dopredu do súboru zachytí komunikáciu s určitým zariadením na ktorej overí, že je didakticky dobrá a ilustruje to čo má. Následne daný súbor posunie študentom aby si mohli zobrazíť analýzu konkrétnych paketov. Užitočná je ale aj analýza priamej interakcie užívateľa s jeho konkrétnym zariadením, preto by sme zároveň chceli podporovať aby si študenti mohli pripojiť vlastné zariadenie a skúmať s ním komunikáciu v reálnom čase. Z toho nám vyplývajú nasledujúce požiadavky:

P2 Mala by byť schopná analyzovať USB pakety zachytené do súboru v rozumnom formáte pomocou predom definovaného snifferu.

P3 Mala by byť schopná analýzy paketov v reálnom čase. To znamená, že bude podporovať čítanie súboru súvisle s tým ako do neho bude zapisovať iný software (za predpokladu, že to daný software povoľuje).

Ako sme mohli vidieť aj na predchádzajúcich príkladoch, hexdump je jednou zo základných funkcií na analýzu paketov. Zároveň sa nám ale nepáčilo, že väčšina hexdumpov je neprehľadná a ťažko sa v nich orientuje. Preto si zdefinujeme nasledujúce požiadavky:

P4 Mala by pomocou hexdumpu vedieť zobraziť dáta, ktoré daný sniffer zachytí a uloží.

P5 Mala by mať prehľadnejší hexdump a užívateľovi uľahčiť orientáciu v ňom. Jednotlivé znaky by mali byť farebne označené na základe ich významu (hlavička paketu, rôzne typy descriptorov a pod.).

K sémantickej analýze sa nám môže hodiť vedieť zobraziť dáta a ich význam pomocou stromovej štruktúry. Pretože sa s našou aplikáciou budeme snažiť vysvetliť základy komunikácie na USB zbernici, mali by sme podporovať sémantickú analýzu všetkých základných USB descriptorov a takisto inputu určitej podmnožiny HID zariadení. Ako posledné by sa nám zišlo vedieť pomocou stromovej štruktúry vyobraziť hlavičku jednotlivých paketov. Z toho celého dostávame nasledovné:

P6 Mala by podporovať sémantickú analýzu (vyobrazenie pomocou stromovej štruktúry) pre všetky základné USB descriptorov spomenuté v USB 2.0 špecifikácii [17] v kapitole 9.6 (ako napríklad *Device descriptor*, *Interface descriptor*, *Endpoint descriptor*, atď.).

P7 Mala by byť schopná pomocou stromovej štruktúry zobraziť sémantický význam dát posielaných danou podmnožinou HID zariadení, do ktorej patrí myš, klávesnica a joystick.

P8 Mala by byť schopná pomocou stromovej štruktúry zobraziť sémantický význam jednotlivých hlavičiek paketov.

Vyššie v texte sme označili funkciu Wiresharku vyobraziť sémantický význam dát na bitovej úrovni (obrázok 1.12) za zaujímavú. Preto by sme ju chceli implementovať aj v našej aplikácii, z čoho vyplýva:

P9 V miestach kde to dáva zmysel, by aplikácia mala byť schopná zobrazovať význam dát až na úrovni jednotlivých bitov.

Nechceme užívateľov hneď zaplaviť všetkými detailnými informáciami o paketoch. Preto by sme mali vedieť zobraziť zopár obecných vecí ku každému paketu a vyobraziť tak pohyb na zbernici, a až v prípade interakcie užívateľa s aplikáciou zobraziť podrobný popis jednotlivých paketov. Zároveň sa nám páčila funkcia Device Monitoring Studio, kde boli jednotlivé pakety farebne rozlíšiteľné, čo zvyšovalo celkový prehľad pohybu paketov na zbernici. Z toho dostávame nasledujúce požiadavky:

P10 Mala by na prvý pohľad jasne zobrazit základné informácie o každom analyzovanom pakete (ako napr. dĺžka paketu, typ prenosu a pod.) a pri bližšom skúmaní jednotlivých paketov detailnejšie zobrazit celú jeho hlavičku. Tieto základné informácie by mali byť farebne rozlišiteľné na základe smeru paketu po zbernici.

P11 Detailnejšie informácie o pakete budú zobrazované na základe interakcie užívateľa s aplikáciou.

Aby sme boli schopní sémantickej analýzy dát myši, klávesnice alebo joysticku podľa osobného výberu užívateľa, musíme si získať informácie o ich inpute z *HID Report Descriptoru*, takže naša ďalšia požiadavka je:

P12 Mala by byť schopná rozparsovať *HID Report Descriptor* takým štýlom, aby bolo neskôr možné sématicky reprezentovať input nami zvolených HID zariadení – myš, klávesnica a joystick.

1.4 Ciele práce

Celkové ciele tejto práce sú nasledovné :

C1 Naprogramovať funkčný analyzátor, ktorý spĺňa všetky požadované funkcie **P1-P12**

C2 Návrh programu musí byť dostatočne obecný aby splňoval nasledujúce:

- Jednoduché rozšírenie o analýzu ďalších typov USB prenosov.
- Jednoduché pridanie sémantickej analýzy pre ďalšie HID zariadenia.

2. USB a Windows

vysvetlenie zakladnych pojmov spojenych USB: historia, usb port/conector, plug and play(<https://docs.microsoft.com/en-us/windows-hardware/drivers/kernel/introduction-to-plug-and-play>), low/full/high speed zariadenia

2.1 USB zbernica

Plug and Play device tree(sposob akym si windows udrziava strom zariadeni na zbernici)(<https://docs.microsoft.com/sk-sk/windows-hardware/drivers/gettingstarted/device-nodes-and-device-stacks>)

2.2 Device object a device stack

PDO,FDO, Device object(<https://docs.microsoft.com/en-us/windows-hardware/drivers/kernel/device-object-to-device-objects>) <https://docs.microsoft.com/en-us/windows-hardware/drivers/kernel/creating-a-device-object>

2.2.1 Drivery

Opisat ako teda bezne analyzatory/sniffery funguju windows driver model(WDM)| : [https://docs.microsoft.com/en-us/windows-hardware/drivers/kernel/types-of-wdm-](https://docs.microsoft.com/en-us/windows-hardware/drivers/kernel/types-of-wdm-drivers) bus driver([https://docs.microsoft.com/en-us/windows-hardware/drivers/kernel/bus-](https://docs.microsoft.com/en-us/windows-hardware/drivers/kernel/bus-drivers) drivers), function driver([https://docs.microsoft.com/en-us/windows-hardware/drivers/kernel/f-](https://docs.microsoft.com/en-us/windows-hardware/drivers/kernel/function-drivers) drivers) a filter driver([https://docs.microsoft.com/en-us/windows-hardware/drivers/kernel/filt-](https://docs.microsoft.com/en-us/windows-hardware/drivers/kernel/filter-drivers) drivers)

2.3 Komunikacia s USB zariadenim

sposob komunikacie operacneho systemu so zariadenim pripojenym na USB zbernicu : IRP([https://docs.microsoft.com/en-us/windows-hardware/drivers/gettingstarted/i-](https://docs.microsoft.com/en-us/windows-hardware/drivers/gettingstarted/irp-overview) o-request-packets) , URB ([https://docs.microsoft.com/en-us/windows-hardware/drivers/usbcon-](https://docs.microsoft.com/en-us/windows-hardware/drivers/usbcon/urb-overview) with-a-usb-device) a pod: <https://docs.microsoft.com/en-us/windows-hardware/drivers/kernel/irps>

2.4 USB descriptory

opis zakladnych USB descriptorov, hlavne tych ktore neskor aj vyuzivam v program(Device, Interface, Endpoint, Configuration, String, Setup) : [https://docs.microsoft.com/en-](https://docs.microsoft.com/en-us/windows-hardware/drivers/usbcon/usb-descriptors) us/windows-hardware/drivers/usbcon/usb-control-transfer

2.4.1 Rozloženie USB zariadenia z hladiska descriptorov

[https://docs.microsoft.com/en-us/windows-hardware/drivers/usbcon/usb-device-](https://docs.microsoft.com/en-us/windows-hardware/drivers/usbcon/usb-device-layout) layout

2.5 HID zariadenia

hid zariadenie obecne, priklady <https://docs.microsoft.com/en-us/windows-hardware/drivers/hid/>

2.5.1 Reporty

Input/Output/Feature reporty.

2.5.2 Report Descriptor

Opis report descriptoru, k comu sluzi, pripadne ako z neho vycitat zaujimave data (neskor vyuzite v programe pri parsovani HID Report Descriptoru na naslednu semanticku analyzu dat ktore posielala zariadenie)

3. Analýza

V tejto kapitole sa pozrieme na zopár problémov a ich riešení, na ktoré sme narazili počas vytvárania nášho analyzátoru. Ako prvé sme sa museli rozhodnúť v akom jazyku budeme náš projekt implementovať. Na to si autor zvolil obecné rozšírený jazyk C++, pretože ho pozná najlepšie a má s ním najviac skúseností. Táto voľba nás zároveň v ničom zásadne neobmedzila a nemala negatívny dopad na celkový výsledok aplikácie. Teraz sa pozrieme na niekoľko konkrétnych problémov a spôsoby akými sme ich vyriešili.

3.1 Získanie USB packetov

Na získavanie USB packetov nám bude obecné slúžiť paket sniffer. Väčšina paket analyzátorov má implementované vlastné sniffery a preto sme sa o to pokúsili tiež. Narazili sme ale na niekoľko zásadných problémov, ktoré sa úzko viažu s platformou na ktorú cieľme s našou aplikáciou – Windows.

Microsoft dokumentácia podrobnejšie opisuje komunikáciu medzi HID zariadením a kernel/user-mode aplikáciou [23]. Pri tejto komunikácii sme schopní zachytiť USB pakety posielané zariadením a neskôr ich analyzovať. Keďže naša aplikácia beží v user-mode, prejdeme si práve tento spôsob komunikácie:

1. Aplikácia nájde a identifikuje HID zariadenie.
2. Aplikácia pomocou metódy *CreateFile* otvorí spojenie s HID zariadením.
3. Aplikácia pomocou *HID API* [24] metód *HidD_XXX* získa *Preparsed Data* a informácie ohľadom HID zariadenia.
4. **Aplikácia použije metódu *ReadFile* resp. *WriteFile* na získanie inputu zariadenia resp. poslanie reportu zariadeniu.**
5. Aplikácia pomocou *HID API* [24] metód *HidP_XXX* interpretuje HID reporty.

Podstatný je práve bod 4 v ktorom vidíme, že pomocou metódy *ReadFile* sme schopní od daného zariadenia získať USB pakety, ktoré reprezentujú jeho input. Tie by sme následne mohli pomocou nášho analyzátoru spracovať. Tu narážame na prvý problém, ktorý sa priamo viaže na platformu Windows a ktorý si detailnejšie opíšeme v nasledujúcej sekcii.

Windows exclusive mód

Windows má definovaný tzv. *Access Mode*, ktorý určuje restrikcii prístupu *HID Klienta* k HID zariadeniu. Ten môže byť buď *Shared* alebo *Exclusive*. *Exclusive Mode* zabraňuje ostatným *HID Klientom* v zachytávaní alebo získavaní inputu HID zariadenia, pokiaľ nie sú hlavným príjemcom daného inputu. Preto z bezpečnostných dôvodov otvára *RIM (Raw Input Manager)* niektoré zariadenia v *Exclusive Mode*.

Ak je zariadenie otvorené v *Exclusive Mode*, aplikácia má stále prístup k niektorým jeho údajom pomocou *HID API* [24] metód *HidD_GetXxx*. Tieto metódy nám obecné umožnia získať niektoré descriptor zariadenia, tak ako aj jeho *Prepared Data*. Nie je nám ale umožnené volať metódu *ReadFile*, takže nemáme akým spôsobom zachytávať komunikáciu HID zariadenia s klientom.

Tabuľka zariadení [25] (obrázok 3.1), ktoré *RIM* otvára v *Exclusive Mode* obsahuje aj tie, ktoré sme si v úvode zvolili ako podmnožinu HID zariadení na analýzu – myš a klávesnica.

Windows supports the following top-level collections:

Usage Page	Usage	Windows 7	Windows 8	Windows 10	Notes	Access Mode
0x0001	0x0001 - 0x0002	Yes	Yes	Yes	Mouse class driver and mapper driver	Exclusive
0x0001	0x0004 - 0x0005	Yes	Yes	Yes	Game Controllers	Shared
0x0001	0x0006 - 0x0007	Yes	Yes	Yes	Keyboard / Keypad class driver and mapper driver	Exclusive

Obr. 3.1: Tabuľka zariadenia ich *Access Mode*. Zariadenia postupne po riadkoch – myš, joystick a klávesnica

Známe knižnice

Keďže komunikáciu nemôžeme zachytávať priamo pomocou *HID API* [24] metód, rozhodli sme sa skúsiť použiť niektoré známe knižnice na sledovanie USB zbernice.

Pravdepodobne najznámejšou z nich je *libusb* [26] – knižnica napísaná v jazyku C, ktorá slúži na poskytnutie prístupu k USB zariadeniu. Je cieľená na programátorov aby im uľahčila vývoj aplikácií, ktoré komunikujú s USB hardwarom. Podporuje viaceré platformy, medzi ktorými sa nachádza aj Windows. Zároveň beží v user-mode, takže aplikácia ktorá ju využíva nepotrebuje žiadne špeciálne privilégia na používanie *libusb API*. Ďalšou výhodou je, že podporuje všetky verzie USB od 1.0 až po 3.1. Je schopná zachytiť všetky typy prenosov (control,bulk,interrupt,isochronus), a teda by sme pomocou nej mohli sledovať celú komunikáciu so zariadením vrátane počiatočného nakonfigurovania. Bohužiaľ, ani pomocou tejto knižnice sme neboli schopní získať prístup k zariadeniam, ktoré Windows otvára v exclusive móde.

Keďže sa zameriavame na HID zariadenia, skúsili sme hľadať niečo viac špecifické práve pre túto triedu USB zariadení, a to nás zaviedlo k *HIDAPI* [27]. *HIDAPI* je ďalšia veľmi rozšírená knižnica, ktorá umožňuje komunikáciu s USB zariadeniami z HID triedy a takisto beží na viacerých platformách vrátane Windowsu. Pretrváva tu ale rovnaký problém ako s minulou knižnicou – nie je možné získať prístup k zariadeniam, ktoré sú otvorené vo Windows exclusive móde.

Vyzerá to tak, že jediné riešenie nášho problému je priamo pracovať s filter driverom, ktorý sa nachádza na driver stacku daného zariadenia.

Driver

Windows ponúka niekoľko vstavaných driverov pre rôzne typy zariadení. Stačí ak nájdeme ten, ktorý nám umožní komunikáciu s konkrétnymi HID zariadeniami, s ktorými bol problém kvôli exclusive módu – klávesnica a myš. Pre tieto dve konkrétne zariadenia momentálne existuje *Moufiltr* [28] (upper-level filter driver pre myš) a *Kbfiltr* [29] (upper-level filter driver pre klávesnicu). Tie ale podporujú len legacy zariadenia – non-USB, non-Bluetooth a non-I2C zariadenia.

Riešením by teda bolo naprogramovanie vlastného filter driveru pre myš a klávesnicu. Toto je ale už nad rámec našej práce a preto na získavanie USB paketov použijeme inú third-party aplikáciu.

Third-party aplikácie

Na zachytávanie USB paketov sme sa rozhodli použiť *USBPcap* [30], s ktorým sme sa už stretli v kapitole 1.2 keď sme opisovali Wireshark a rôzne sniffery, s ktorými je schopný spolupracovať. USBPcap je veľmi rozšírený Windows sniffer USB paketov a jeho hlavnou výhodou je jednoduchá inštalácia. Takisto nám vyhovuje, že je s ním schopný spolupracovať Wireshark, čo nám častokrát počas vývoja poskytlo spôsob akým sme si mohli overiť správnosť nami vykonanej analýzy niektorých paketov. Daný sniffer zachytáva pakety do súborov v obecné známom formáte – pcap [31].

Miernou nevýhodou je, že nepodporuje čítanie zo súboru do ktorého práve zapisuje – analýzu paketov v reálnom čase tak budeme musieť vykonať trochu iným spôsobom, a to za pomoci Wiresharku. Ako sme už spomínali, Wireshark je schopný spolupráce s USBPcapom a zároveň podporuje ukladať výstup analýzy do súboru, ktorý je takisto formátu pcap. Wireshark už ale podporuje čítanie z daného súboru počas toho, ako do neho zapisuje. To znamená, že ak budeme chcieť vykonať analýzu v reálnom čase, urobíme to pomocou Wiresharku (konkrétnejší postup si ukážeme neskôr v užívateľskej dokumentácii v kapitole 6).

3.2 Sémantická analýza dát

Ako sme už naznačili v kapitole 1.1, na vykonanie sémantickej analýzy inputu HID zariadenia je potrebné získať informácie o danom inpute z Report Descriptoru konkrétného zariadenia. V tejto kapitole si vysvetlíme formát Report Descriptoru a ako nám informácie ktoré reprezentuje pomôžu v sémantickej analýze inputu HID zariadenia.

Report Descriptor

Celková štruktúra Report Descriptoru je opísaná v HID Class Specification [32]. Descriptor sa skladá z tzv. itemov, ktoré obsahujú informácie o danom zariadení. Každý item obsahuje hlavičku, ktorá sa skladá z troch častí:

- Typ
- Tag
- Veľkosť

Existujú 3 typy itemov : *Main*, *Global* a *Local*. Main item udáva informácie o konkrétnej časti zariadenia (napríklad o tlačidlách na klávesnici, alebo o hodnotách osy X a Y pri pohybe myšou, atď.) Global a Local itemy bližšie špecifikujú Main item (napríklad počet bytov, ktorý je potrebný na reprezentovanie jednotlivých súradníc pri pohybe myšou). Local item určuje vlastnosti len najbližšieho Main itemu, zatiaľ čo Global item je platný pre všetky nasledujúce Main itemy.

Main item má momentálne definovaných 5 tagov:

- Input – udáva informácie o inpute zariadenia. Toto je veľmi podstatná časť, pretože práve na základe tohto itemu vieme, že v inpute zariadenia máme očakávať byty reprezentujúce konkrétnu časť zariadenia (bližšie informácie o konkrétnej časti inputu, ako napríklad veľkosť a význam, sú poskytnuté Global a Local itemami pred týmto konkrétnym Input itemom)
- Output – podobný význam ako Input item, ale reprezentuje dáta, ktoré sú posielané zariadeniu (napríklad nastavenie LED svetiel na klávesnici)
- Feature – itemy ktoré môžu byť posielané ako input aj output a využívajú sa napríklad na kalibráciu konkrétneho zariadenia.
- Collection – udáva akúsi logickú kolekciu/zoskupenie rôznych Input/Output/Feature itemov. Napríklad kolekcia myši by mohla obsahovať Input itemy reprezentujúce tlačidlá a súradnice pohybu myši.
- End Collection – udáva koniec kolekcie Collection itemu.

Vysvetlíme si ešte zopár hlavných Local/Global tagov, ktoré budeme potrebovať aby sme pochopili Report Descriptor:

- Usage – reprezentuje odporúčaný význam dát (napríklad, že sa jedná o tlačidlo, osy X/Y/Z, atď.), ale každý výrobca si ho môže definovať po svojom.
- Usage Page – zoskupenie jednotlivých Usage-ov do rôznych tried (napríklad Generic Desktop Page, Keyboard Page, LED Page, atď.). Kompletný výpis je dostupný v HID Usage Tables špecifikácii [33]. To znamená, že Usage s hodnotou 0x00 môže mať iný význam v Generic Desktop Page ako v Keyboard Page.
- Report Size a Report Count – reprezentujú veľkosť dát. Napríklad ak je Report Size = 8 a Report Count = 1, dáta budú reprezentované jednou 8-bitovou hodnotou, takže 1 byte.
- Logical Minimum a Logical Maximum – označujú rozsah hodnôt aké môžu dáta nadobúdať.

Momentálne keď už máme celkový prehľad o základných itemoch, môžeme si ukázať parsovanie Report Descriptoru na konkrétnom príklade. Na obrázku 3.2 môžeme vidieť konkrétny Report Descriptor prevzatý z HID Class Specification [34], reprezentujúci myš. Ten si teraz postupne rozoberieme.

```

Usage Page (Generic Desktop),
Usage (Mouse),
Collection (Application),
    Usage (Pointer),
    Collection (Physical),
        Report Count (3),
        Report Size (1),
        Usage Page (Buttons),
        Usage Minimum (1),
        Usage Maximum (3),
        Logical Minimum (0),
        Logical Maximum (1),
        Input (Data, Variable, Absolute),
        Report Count (1),
        Report Size (5),
        Input (Constant),
        Report Size (8),
        Report Count (2),
        Usage Page (Generic Desktop),
        Usage (X),
        Usage (Y),
        Logical Minimum (-127),
        Logical Maximum (127),
        Input (Data, Variable, Relative),
    End Collection,
End Collection

```

Obr. 3.2: Konkrétny príklad Report Descriptoru myši prevzatý z HID Class Specification [34]

Z prvej časti (obrázok 3.3) si môžeme všimnúť, že sa celý Report Descriptor delí do logických kolekcí – najprv *Application* a potom *Physical*. Dôležitá je takisto Usage informácia, kde je hneď na začiatku ohraničená Usage Page na *Generic Desktop* a samotný *Usage (Mouse)* nám bližšie špecifikuje, že sa jedná o myš.

```

Usage Page (Generic Desktop),
Usage (Mouse),
Collection (Application),
    Usage (Pointer),
    Collection (Physical),

```

Obr. 3.3: Časť Report Descriptoru reprezentujúca jednotlivé kolekcie

Na obrázku 3.4 vidíme ďalšiu časť descriptoru, ktorá reprezentuje tlačidlá myši – to sme vyčítali z časti *Usage (Buttons)*. Nasledujúce časti *Report Count (3)* a *Report Size (1)* nám naznačujú, že veľkosť istej podmnožiny dát budú tri 1-bitové hodnoty. Ďalej z hodnôt *Logical Minimum (0)* a *Logical Maximum (1)* vidíme že budú nabývať hodnoty medzi 0 a 1. Nasleduje Input item s hodnotami *Data*, *Variable* a *Absolute* – tie nám udávajú, že sa jedná o input zariadenia, ktorý bude modifikovateľný (hodnota *Data*), budú to 3 jednotlivé 1-bitové položky (hodnota *Variable*) a hodnota dát nie je relatívna voči inej hodnote (položka *Absolute*). Ďalej vidíme časti *Report Count (1)*, *Report Size (5)* a *Input (Constant)* – Takže v inpute bude nasledovať časť o veľkosti 5 bitov, ktorá bude mať konštantnú readonly hodnotu (toto v myši väčšinou reprezentuje padding tlačidiel).

```
Report Count (3),
Report Size (1),
Usage Page (Buttons),
Usage Minimum (1),
Usage Maximum (3),
Logical Minimum (0),
Logical Maximum (1),
Input (Data, Variable, Absolute),
Report Count (1),
Report Size (5),
Input (Constant),
```

Obr. 3.4: Časť Report Descriptoru reprezentujúca tlačidlá myši

Z nasledujúcej časti (obrázok 3.5) vidíme, že sa bude jednať o dve 8-bitové hodnoty (položky *Report Size (8)* a *Report Count (2)*). Následne prepne Usage Page z *Buttons* na *Generic Desktop* a definujeme 2 Usage pre *X* a *Y* – tie v myši reprezentujú jednotlivé osy pohybu. Ďalej vidíme, že môžu nadobúdať hodnoty od -127 do 127 (položky *Logical Minimum (-127)* a *Logical Maximum (127)*) čo presne odpovedá rozsahu jedného znamienkového bytu.

```
Report Size (8),
Report Count (2),
Usage Page (Generic Desktop),
Usage (X),
Usage (Y),
Logical Minimum (-127),
Logical Maximum (127),
Input (Data, Variable, Relative),
```

Obr. 3.5: Časť Report Descriptoru reprezentujúca osy X a Y pri pohybe s myšou

Na poslednom obrázku 3.6 už len vidíme ukončenie kolekcií, tak ako aj celého Report Descriptoru.

End Collection,
End Collection

Obr. 3.6: Časť Report Descriptoru reprezentujúca ukončenie jednotlivých kolekcií

V konečnom dôsledku nám z toho teda vychádza, že input našej myši bude mať veľkosť 3 byty (sčítanie všetkých $Report\ Size * Report\ Count$), kde prvé 3 bity budú reprezentovať stlačenie tlačidiel, nasledujúcich 5 bitov bude výplň a posledné 2 byty reprezentujú osy X a Y v tomto poradí. Toto reprezentuje jeden input report nášho zariadenia na jeho konkrétnom endpointe. Jeden endpoint ale môže obsahovať viac reportov (či už input, output alebo feature). V takom prípade musia byť jednotlivé reporty prefixované 1B hodnotou, ktorú nazývame *Report ID* a ktorá musí byť uvedená v Report Descriptore pri jednotlivých reportoch v podobe *Report ID* itemu. Ako príklad niečoho takého si môžeme predstaviť klávesnicu so zabudovaným zariadením s funkciou myši. Takáto klávesnica môže posilať informácie o stlačení kláves tak ako aj o pohybe myši v jednom endpointe. Na rozlíšenie o ktoré dáta sa jedná by sa použil Report ID.

Ostáva nám ešte jedna možnosť, ktorú sme doteraz nespomenuli – ako priradiť Report Descriptor k jednotlivým reportom pokiaľ ich zariadenie posila z viacerých endpointov. To si vysvetlíme v nasledujúcej sekcii.

Report Descriptor \longleftrightarrow endpoint

Ako už vieme z kapitoly (ODKAZ), jedno USB zariadenie môže pozostávať z viacerých interfacov. Každý interface má dané endpointy, ktoré sa s ním viažu. Pri komunikácii medzi USB hostom a zariadením si z hlavičky jednotlivých paketov vieme vytiahnuť informáciu do ktorého endpointu zapisujeme, resp. z ktorého endpointu čítame dáta. Takže v prípade reportu poslaného zariadením vieme zistiť k akému endpointu sa viaže. Hodilo by sa nám teda vedieť k jednotlivým endpointom priradiť im odpovedajúci Report Descriptor. Túto informáciu máme nepriamo poskytnutú v Setup Pakete, ktorý posila USB host v momente keď si od zariadenia vypýta jeho Report Descriptor. Špecifikuje v ňom totiž položku *wInterfaceNumber*, ktorá udáva číslo interfacu pre ktorý si USB host vypýtal Report Descriptor. Teraz nám už len stačí zistiť aké endpointy sa na daný interface viažu – túto informáciu sme získali vyššie v konfigurácii keď si USB host vypýtal od zariadenia Configuration Descriptor a spolu s ním dostal aj Interface Descriptor a jemu odpovedajúce Endpoint Descriptor.

3.3 Voľba frameworku

Pre jazyk C++ existuje hneď niekoľko známych GUI frameworkov nad ktorými sme uvažovali, ako napríklad *Dear ImGui* [35], *SFGui* [36] alebo *Qt* [37]. Autor tohto projektu nemal so žiadnym z týchto frameworkov predchádzajúcu skúsenosť, ale na základe rešeršu si nakoniec zvolil Qt.

Qt

Qt je open-source framework na vytváranie užívateľských rozhraní. Podporuje všetky základné platformy ako Windows, Linux a macOS. Je to jeden z najznámejších a najpoužívanějších GUI frameworkov v profesionálnom prostredí medzi firmami ako napríklad LG, AMD, Panasonic a podobne. To naznačuje, že sa jedná o robustný systém s rozsiahlou a dobre otestovanou funkcionalitou. Qt sa môže na prvý pohľad javiť ako zložitejší na naučenie kvôli jeho komplexnosti, na druhú stranu sa len ťažko hľadá niečo, čoho by nebol schopný. To nás dostáva k trochu špecifickému spôsobu akým sa v Qt programuje, a to je využívaním tzv. model/view architektúry [38].

Model/View Architektúra

Qt obsahuje množinu itemov, ktoré používajú model/view architektúru na riadenie vzťahu medzi reprezentáciou dát a spôsobom akým sú vyobrazené. Celá architektúra vychádza zo známeho návrhového vzoru MVC [39] (Model-View-Controller), kde sa View a Controller zlúčia do jedného. V prípade ak chceme mierne upraviť spôsob akým viewer dáta zobrazuje (napríklad zmeniť ich farbu a podobne) použijeme na to tzv. *delegate*. Architektúra sa dá teda rozdeliť do 3 hlavných komponent – model, view a delegate. Každá z týchto komponent je definovaná abstraktnou triedou, ktoré ponúkajú jednotný interface. V niektorých prípadoch (napríklad `QFileSystemModel` [40], `QTableView` [41], atď.) máme dokonca už predimplementovanú aj základnú funkcionality. Komunikácia medzi komponentami prebieha pomocou tzv. *signals* a *slots* [42]. Signal je vyvolaný nejakým konkrétnym eventom (napríklad stlačenie tlačidla na konkrétny item) a slot je funkcia, ktorá sa zavolá ako odpoveď na daný signal.

Qt má ešte jednu obrovskú výhodu, a tou je rozsiahla komunita a skvelá dokumentácia. V prípade akýchkoľvek otázok alebo nejasností sme nikdy nemali problém nájsť odpoveď, či už v spomínanej dokumentácii, alebo na rôznych užívateľských fórach.

3.4 Spracovávanie pcap súborov

Pcap je obecné známy formát súboru so špecifickým zameraním – ukladanie dát, ktoré majú reprezentovať pakety. Existuje niekoľko spôsobov ako daný súbor spracovať:

Npcap API

Npcap [43] je Windows knižnica na zachytávanie sieťových paketov, ktorá poskytuje Npcap API [44]. Pomocou tohto API sme schopní spracovávať pcap súbory. Schopnosť Npcap zachytávať pakety je nám zbytočná – jednak pretože na to využívame USBPcap a navyše Npcap je schopný zachytiť iba sieťové pakety. Jej API by nám ale mohlo zjednodušiť čítanie nami zachytených pcap súborov. Pomocou volania `pcap_open_offline()` získame handle pomocou ktorého môžeme neskôr čítať jednotlivé pakety volaním funkcie `pcap_next()`, ktorá nám vráti pointer na dáta samotného paketu. Použitie je skutočne jednoduché, má to ale aj svoje nevýhody.

Za prvú nevýhodu môžeme brať nutnosť pridávať do projektu ďalšie knižnice – čo zahŕňa inštaláciu Npcap, linkovanie dll súborov do nášho projektu a celkovú integráciu nášho projektu s knižnicou. Zároveň tým vytvárame akúsi závislosť nášho projektu na Npcap knižnici. Čo ale môžeme považovať za najväčšiu nevýhodu je fakt, že použitím tejto knižnice úzko viažeme náš program na platformu Windows. Windows je síce platforma na ktorú hlavne cieľime, ale zároveň môžeme uvažovať nad neskôrším rozšírením na iné platformy. V takom prípade by sme museli prepísať dôležitú časť nášho kódu, ktorá by zahŕňala spracovanie pcap súborov a oddelenie dát jednotlivých paketov.

Keď sa bližšie pozrieme na formát pcap súboru, zistíme, že vôbec nie je zložitý. Celý súbor obsahuje globálnu hlavičku a dáta jednotlivých paketov sú prefixované lokálnymi hlavičkami. Preto spracovanie jednotlivých súborov nie je náročné a môžeme na to použiť bežné spôsoby.

QFile

Bežný prístup k čítaniu súborov v C++ by bolo použitie *std::ifstream* [45], keďže ale používame Qt framework, máme k dispozícii triedu *QFile* [46]. *QFile* ponúka jednoduché API vďaka ktorému môžeme postupne spracovávať pcap súbor. Ako sme už spomínali vyššie v kapitole 3.1, na analýzu paketov v reálnom čase používame ukladanie do súboru pomocou Wiresharku. *QFile* API nám umožňuje tento súbor postupne spracovávať a tým pádom sme splnili požiadavky **P2** a **P3** ohľadom čítania súborov, ktoré sme si na začiatku stanovili.

3.5 Uchovávanie informácií

Počas spracovávania pcap súborov máme k dispozícii dáta popisujúce jednotlivé pakety. Tie si musíme nejakým spôsobom zarepresentovať a ukladať, aby sme ich mohli neskôr analyzovať a vyobraziť. V tejto sekcii si rozoberieme niekoľko možných riešení.

char[]

Metóda *read(char*, qint64)* pomocou ktorej čítame zo súboru špecifikuje prvým parametrom miesto do ktorého sa majú zapísať prečítané dáta a druhým parametrom akú veľkosť dát chceme prečítať. Musíme si teda dopredu pripraviť buffer do ktorého budeme zapisovať. Keďže konkrétnu veľkosť jednotlivých paketov nepoznáme, budeme musieť buffer dynamicky alokovať na halde. Na to existujú 2 hlavné spôsoby ako niečo také dosiahnuť:

- Pomocou *new char[paket_size]* – tento spôsob by sa dal v dnešnom C++ považovať za zastaralý a nebezpečný. Je nutné pečlivo sledovať životnosť takto alokovanej pamäte, musíme si na ňu držať odkaz a keď už ju nebudeme potrebovať, je potrebné na nej zavolať delete aby nám nevznikali žiadne memory leaky.
- Pomocou *std::unique_ptr<char[]>* – tento spôsob je určite lepší ako ten predchádzajúci, pretože nám odpadá starosť o memory leaky. Na druhú stranu tu vzniká menší problém z uskladnením alebo s predávaním dát

po programe v podobe parametru. Museli by sme sa starať o to, kto je vlastníkom daného objektu a či je ešte korektné ho používať.

Skúsime sa pozrieť na to, čo nám ponúka samotné Qt.

QByteArray

V riešení nám zase pomôže koncept dostupný v Qt – *QByteArray* [47]. *QByteArray* je classa ktorá slúži práve na uchovávanie bytov alebo '0'-terminated stringov. Navyše sú uložené dáta copy-on-write takže ušetríme pamäť a zbytočné kopírovanie dát. Obsahuje metódy ako *resize(int)* (zmení veľkosť daného poľa) alebo *data()* (vráti char* na začiatok poľa). O životnosť objektu sa nemusíme obávať, pretože Qt používa vlastný reference counting systém, ktorým sa stará o všetky zdieľané objekty medzi ktoré patrí aj *QByteArray*. O ukladanie dát sa budeme bližšie baviť v sekcii 3.6 nižšie. Aby sme to ale pochopili, vysvetlíme si tu ešte jeden koncept Qt, ktorý neskôr využijeme spolu s *QByteArray* pri ukladaní dát – *QVariant* [48]. *QVariant* je trieda, ktorá sa správa ako union pre väčšinu základných Qt datových typov. Podstatné je, že si doňho vieme zabaliť náš konkrétny *QByteArray* a neskôr ho odtiaľ získať zavolaním metódy *toByteArray()*.

3.6 Zobrazenie základných informácií

V požiadavke **P10** sme si zdefinovali aby aplikácia na prvý pohľad zobrazila len základné údaje o pakete, a detailnejšie informácie by mali byť podľa požiadavky **P11** zobrazené až po interakcii užívateľa. Pravdepodobne najintuitívnejšia interakcia užívateľa ak chce zobraziť detailnejšie informácie je pomocou dvojkliku na konkrétnu položku. Zobrazenie základných informácií by malo byť podobné tým, aké sme mali možnosť vidieť u Wiresharku a Device Monitoring Studia v kapitole 1.2. Qt nám poskytuje 2 základné classy pomocou ktorých budeme schopní dosiahnuť podobného vzhľadu.

QListWidget

QListWidget [49] poskytuje vzhľad na základe view classy *QListView* [50] a jednotlivé položky sú reprezentované pomocou itemu *QListWidgetItem* [51]. Pomocou *QListWidget* sme boli schopní dosiahnuť výsledku ukázaného na obrázku 3.7.

QListWidget zároveň podporuje signal *itemDoubleClicked(QListWidgetItem*)*, ktorý má ako parameter konkrétny *QListWidgetItem* na ktorý užívateľ dvojklikol. Keďže chceme po dvojkliku zobraziť detailnejšie informácie o danom pakete, musíme si ich najprv niekde uschovať a následne sa k nim vedieť ľahko dostať. Tu prichádza výhoda použitia *QListWidgetItem* – ten je totiž schopný si v sebe uchovať dáta v podobe *QVariantu*. To nám úplne vyhovuje, pretože ako sme už spomínali vyššie v sekcii 3.5, dáta si uchováваме práve v *QByteArray*, ktorý sme schopný zabaliť do *QVariantu*. Uloženie konkrétneho *QVariantu* je vykonané pomocou metódy *setData(int role, const QVariant &value)* – role môže byť podľa Qt dokumentácie [52] hodnota 256 a vyššie, druhý parameter je práve *QVariant*,

0	host	6.0	36	CONTROL TRANSFER	DEVICE_DESCRIPTOR
1	6.0	host	46	CONTROL TRANSFER	
2	host	6.0	36	CONTROL TRANSFER	CONFIGURATION_DESCRIPTOR
3	6.0	host	37	CONTROL TRANSFER	
4	host	6.0	36	CONTROL TRANSFER	CONFIGURATION_DESCRIPTOR
5	6.0	host	80	CONTROL TRANSFER	
6	host	6.0	36	CONTROL TRANSFER	
7	6.0	host	28	CONTROL TRANSFER	
8	host	6.0	27	UNKNOWN TRANSFER	
9	6.0	host	27	UNKNOWN TRANSFER	
10	host	6.0	27	UNKNOWN TRANSFER	
11	6.0	host	27	UNKNOWN TRANSFER	
12	host	6.0	36	CONTROL TRANSFER	STRING_DESCRIPTOR
13	6.0	host	32	CONTROL TRANSFER	
14	host	6.0	36	CONTROL TRANSFER	STRING_DESCRIPTOR
15	6.0	host	52	CONTROL TRANSFER	
16	host	6.0	36	CONTROL TRANSFER	STRING_DESCRIPTOR
17	6.0	host	32	CONTROL TRANSFER	
18	host	6.0	36	CONTROL TRANSFER	STRING_DESCRIPTOR
19	6.0	host	32	CONTROL TRANSFER	

Obr. 3.7: Vyobrazenie základných informácií o pakete pomocou QListWidget

ktorý chceme uložiť. Odkázaním sa na rovnakú *role* hodnotu v metóde *data(int role)* dostaneme ako návratovú hodnotu QVariant ktorý sme si vyššie uložili. Ďalšou výhodou je, že jeden riadok reprezentuje práve jeden QListWidgetItem, takže presne vieme v ktorom iteme máme uložené dáta.

Nevýhody sú naopak nepraktické rozšírenie o ďalšie stĺpčky. Tým, že jeden riadok je práve jeden item, stĺpčky sú tu len akási ilúzia, pretože všetky informácie ktoré môžeme vidieť na obrázku 3.7 (napríklad index paketu na začiatku alebo typ prenosu), je len jeden dlhý string. Takisto tu tým pádom nemáme žiadnu zmysluplnú hlavičku, ktorá by nám udávala význam jednotlivých „stĺpčekov“.

QTableWidget

QTableWidget [53] poskytuje vzhľad na základe view classy QTableView [41] a jednotlivé položky sú reprezentované pomocou itemu QTableWidgetItem [54]. Pomocou QTableWidget sme dosiahli výsledok vyobrazený na obrázku 3.8 nižšie.

Index	Source	Destination	Length	Transfer type	Function
0	host	6.0	36	CONTROL TRANSFER	DEVICE_DESCRIPTOR
1	6.0	host	46	CONTROL TRANSFER	
2	host	6.0	36	CONTROL TRANSFER	CONFIGURATION_DESCRIPTOR
3	6.0	host	37	CONTROL TRANSFER	
4	host	6.0	36	CONTROL TRANSFER	CONFIGURATION_DESCRIPTOR
5	6.0	host	80	CONTROL TRANSFER	
6	host	6.0	36	CONTROL TRANSFER	
7	6.0	host	28	CONTROL TRANSFER	
8	host	6.0	27	UNKNOWN TRANSFER	
9	6.0	host	27	UNKNOWN TRANSFER	
10	host	6.0	27	UNKNOWN TRANSFER	
11	6.0	host	27	UNKNOWN TRANSFER	
12	host	6.0	36	CONTROL TRANSFER	STRING_DESCRIPTOR
13	6.0	host	32	CONTROL TRANSFER	
14	host	6.0	36	CONTROL TRANSFER	STRING_DESCRIPTOR
15	6.0	host	32	CONTROL TRANSFER	

Obr. 3.8: Vyobrazenie základných informácií o pakete pomocou QTableWidget

Ako si môžeme všimnúť, podoba medzi `QListWidget` a `QTableWidget` tak ako aj medzi `QListWidgetItem` a `QTableWidgetItem` je zrejmá, a teda nie je prekvapivé, že budú mať niektoré interné vlastnosti rovnaké. Medzi ne patria našťastie aj metódy `setData(int role, const QVariant &value)` a `data(int role)` – takže výhoda uchovania si dát nám ostala. `QTableWidget` takisto podporuje signal `itemDoubleClicked(QTableWidgetItem*)` s rovnakými vlastnosťami ako mal `QListWidget`. Naopak rozdiel je napríklad v tom, že jeden riadok je teraz reprezentovaný niekoľkými `QTableWidgetItem`mi. To nám umožňuje pridať si samotný item pre každú informáciu o pakete, ktorú chceme vyobraziť, čím zároveň vytvárame skutočné stĺpčeky. Pridanie nového stĺpčeka je teda jednoduché – pridáme len nový item v riadku. `QTableWidget` má zároveň možnosť pridania horizontálnej hlavičky, v ktorej vieme špecifikovať význam hodnôt v jednotlivých stĺpčkoch.

S týmito vlastnosťami nám ale vznikli aj nové problémy, ktoré musíme riešiť. Prvý problém je, že nie všetky riadky majú jednotnú dĺžku (niektoré nemajú vyplnenú hodnotu „Function“, ako môžeme vidieť na obrázku 3.8 vyššie). To znamená, že riadok tam pôvodne nemá žiaden item – pri dvojkliku na nevyplnenú časť by sa teda nič nestalo. Takisto pri kliknutí na riadok alebo pri vyfarbení daného riadku by bola označená resp. vyfarbená len časť v ktorej sa nachádza text – to pôsobí nekonzistentným dojmom. Riešenie je našťastie triviálne – v prípade, že do daného stĺpčeka nemáme čo vyplniť, vytvoríme prázdny `QTableWidgetItem` a ten pridáme. Ďalší problém je uchovávanie dát – v prípade `QListWidget` bol jeden riadok reprezentovaný len jedným itemom, takže v prípade keď užívateľ dvojklikol na paket, ktorý chcel bližšie preskúmať, tak sme v signali `itemDoubleClicked(QListWidgetItem*)` dostali práve item v ktorom sme mali uložené všetky informácie. V prípade `QTableWidget` ak užívateľ dvojklikne na určitú časť riadku, zvolí sa jeden konkrétny item spomedzi všetkých na danom riadku. Problém teraz spočíva v tom, v ktorom iteme si budeme uchovávať dáta. Trochu prvoplánové riešenie by bolo ukladať si dáta do každého itemu v riadku – toto riešenie nie je príliš ideálne, pretože vzniká zbytočná duplikácia dát. Preto si dáta uchováваме v prvom iteme každého riadku a v prípade keď užívateľ dvojklikom vyberie konkrétny item, zistíme si pomocou metódy `row(QTableWidgetItem*)` číslo riadku na ktorom sa nachádza a následne pomocou metódy `item(int row, int column)` získame prvý item v danom riadku.

V požiadavke **P10** sme si takisto určili, že chceme jednotlivé pakety mať farebne oddelené na základe ich pohybu na zbernici. Túto informáciu získame z hlavičky paketu pri vytváraní jednotlivých riadkov. Rozhodli sme sa vyfarbovať riadky ktoré reprezentujú pakety posielané zariadením hostovi, pretože sa v nich väčšinou nachádzajú zaujímavejšie dáta a myslíme si, že typický užívateľ má tendenciu klikať skôr na farebne zvýraznené časti. Keďže tu zase nemáme jeden item reprezentujúci celý riadok, zafarbenie prebieha obyčajnou iteráciou cez všetky stĺpčeky a ich samostatným zafarbením. Celkovú funkcionálnosť farebného oddelenia sme rozvinuli ešte o trochu viac oproti ostatným aplikáciám a dokonca ešte aj farba akou je item označený, sa líši na základe typu prenosu. Výsledný vzhľad môžeme vidieť na obrázku 3.9

Index	Source	Destination	Length	Transfer type	Function
21	6.0	host	28	CONTROL TRANSFER	
22	host	6.0	36	CONTROL TRANSFER	HID_REPORT_DESCRIPTOR
23	6.0	host	84	CONTROL TRANSFER	
24	host	6.1	27	INTERRUPT TRANSFER	
25	host	6.1	27	INTERRUPT TRANSFER	
26	host	6.0	36	CONTROL TRANSFER	
27	6.0	host	28	CONTROL TRANSFER	
28	host	6.0	36	CONTROL TRANSFER	HID_REPORT_DESCRIPTOR
29	6.0	host	50	CONTROL TRANSFER	
30	host	6.0	36	CONTROL TRANSFER	STRING_DESCRIPTOR
31	6.0	host	30	CONTROL TRANSFER	
32	6.1	host	31	INTERRUPT TRANSFER	
33	host	6.1	27	INTERRUPT TRANSFER	
34	6.1	host	31	INTERRUPT TRANSFER	
35	host	6.1	27	INTERRUPT TRANSFER	

Obr. 3.9: Vyobrazenie základných informácií o pakete pomocou QTableWidgetItem

3.7 Hexdump

Ako sme si stanovili na začiatku v požiadavke **P4**, chceme aby naša aplikácia bola schopná zobrazovať hexdump dát nad ktorými prebieha analýza. Chceli by sme podporovať základné správanie hexdumpu – označením hexovej časti sa automaticky označia im odpovedajúce tlačiteľné znaky a opačne. Zároveň ale máme na náš hexdump vďaka požiadavke **P5** trochu väčšie nároky, a to farebné vyznačenie jednotlivých znakov podľa významu. V Qt máme 2 základné možnosti ako niečoho takého dosiahnuť – nadefinovanie vlastného widgetu alebo prispôsobenie už existujúceho widgetu naším požiadavkám. Poďme sa teda pozrieť na to čo všetko by sme museli riešiť v prípade nadefinovania vlastného widgetu.

QAbstractScrollArea

QAbstractScrollArea [55] class je v podstate widget, ktorý nám poskytuje abstrakciu scrollovacej plochy (scrollovanie samozrejme musíme podporovať v prípade väčšieho množstva dát). Kontent je vykresľovaný do widgetu, ktorý nazývame viewport. Dokumentácia udáva list niekoľkých vecí o ktoré sa musíme postarať ak chceme objekt, ktorý dedí od QAbstractScrollArea:

- Nastavenie scroll-barov – to zahŕňa napríklad ich veľkosť, rozsah pohybu, definovanie o koľko sa má daný scroll-bar pohnúť v prípade ak užívateľ stlačí tlačidlo „Page Up“ resp. „Page Down“, alebo aj sledovanie ich aktuálnej pozície.
- Vykreslenie kontentu, ktorý sa má zobrazíť na základe hodnôt jednotlivých scroll-barov (vykresliť tú časť hexdumpu na ktorej sa momentálne podľa scroll-barov nachádzame).
- Spracovať všetky eventy, ktoré sú prijaté viewportom – napríklad paintEvent (požaduje vykreslenie celého viewportu alebo danej časti), mousePres-

sEvent (kliknutie na viewport), atď.

- Používanie *viewport->update* k updatenutiu obsahu na viewporte namiesto *update()*.

Ako vidíme, je toho celkom dosť a preto skúsime nájsť widget, ktorý už má v Qt všetky tieto požiadavky vyriešené a len ho upravíme našim požiadavkám hexdumpu.

QTableView

S QTableView [41] sme sa už stretli vyššie v sekcii 3.6. Hexdump si v podstate vieme predstaviť ako tabuľku dát, preto dáva zmysel skúsiť upraviť práve tento view. QTableView je súčasťou Qt model/view frameworku, takže nám umožňuje logicky oddeliť dáta a ich zobrazovanie. Na reprezentáciu dát máme vytvorený vlastný model, podme sa preto venovať len ich zobrazeniu pomocou table view. Na kompletný hexdump budeme potrebovať 2 samostatné QTableView – jeden pre hexa časť a jeden pre tlačiteľné znaky. Pre vyobrazenie hexdumpu tak, aby sa podobal tomu na čo je bežný užívateľ zvyknutý nám postačia len jednoduché úpravy ako napríklad:

- Odstránenie horizontálnej hlavičky (*horizontalHeader()->setVisible(false)*).
- Nastavenie veľkosti jednotlivých buniek (*resizeColumnsToContents()* a *resizeRowsToContents()*).
- Odstránenie mriežky medzi bunkami (*setShowGrid(false)*).

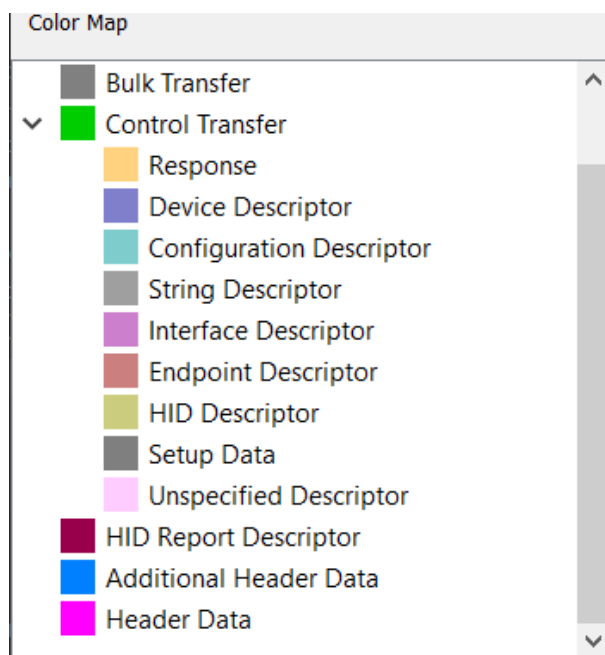
Na implementáciu funkcionality označenia rovnakých častí tlačiteľných znakov ako hexdumpu a opačne nám stačí prepojiť signály *selectionChanged* jednotlivých selection modelov pre oba QTableView s nami definovanými funkciami (bližšie si ich opíšeme v kapitole Vývojová dokumentácia 4). Keďže sa pohybujeme v model/view frameworku, na vykreslenie jednotlivých dát použijeme delegate, ktorý sa takisto stará o ich farebné oddelenie (detailnejší popis jeho fungovania si takisto opíšeme v kapitole Vývojová dokumentácia 4). Ukážku nášho výsledného hexdumpu môžeme vidieť na obrázku ?? nižšie.

Address	Hexadecimal values	Printable
000000	1C 00 A0 49 1F 6A 8A DC FF FF 00 00 00 00 08 00 l . j
000010	01 01 00 06 00 80 02 34 00 00 00 03 09 02 34 00 4
000020	02 01 00 A0 32 09 04 00 00 01 03 01 02 00 09 21 2
000030	11 01 00 01 22 38 00 07 05 81 03 04 00 0A 09 04 " 8
000040	01 00 00 03 00 00 00 09 21 11 01 00 01 22 16 00

Obr. 3.10: Vyobrazenie hexdumpu s farebným oddelením dát podľa významu

Samozrejme, obyčajné vyfarbenie nám k lepšej orientácii v hexdumpe veľmi nepomôže, pokiaľ nepoznáme význam jednotlivých farieb. Preto sme implementovali tzv. Color Map, ktorý nám k jednotlivým farbám priradí ich význam. Jej časť môžeme vidieť na obrázku 3.11.

Takisto si uvedomujeme, že nie každému môže farebné zvýraznenie hexdumpu vyhovovať a preto dávame užívateľovi možnosť ho úplne vypnúť. Výsledok je ukázaný na obrázku 3.12.



Obr. 3.11: Časť Color Map

Address	Hexadecimal values	Printable
000000	1C 00 A0 49 1F 6A 8A DC FF FF 00 00 00 00 08 00 j
000010	01 01 00 06 00 80 02 34 00 00 00 03 09 02 34 00 4 4 .
000020	02 01 00 A0 32 09 04 00 00 01 03 01 02 00 09 21 2 !
000030	11 01 00 01 22 38 00 07 05 81 03 04 00 0A 09 04 " 8
000040	01 00 00 03 00 00 00 09 21 11 01 00 01 22 16 00 ! "

Obr. 3.12: Vyobrazenie hexdumpu bez farebného oddelenia dát podľa významu

3.8 Zobrazenie sémantického významu dát

Ako sme si definovali v požiadavkách **P6- P8**, chceme byť schopní vykonávať sémantickú analýzu pomocou stromovej štruktúry. Vzhľadom na to, že sa jedná o obecné známy spôsob vyobrazenia dát, Qt má na to špeciálnu classu `QTreeView` [56], ktorá je súčasťou Qt model/view frameworku. Opäť ale máme možnosť nadefinovať si vlastný view pomocou dedenia od classy `QAbstractItemView` [57]. Ako sme už ale videli vyššie, museli by sme sami implementovať viaceré funkcionality, ktoré nám už `QTreeView` ponúka sám a preto nevidíme dôvod nadefinovania vlastného view.

QTreeView

Celkom používame až 3 `QTreeView` – pre vyobrazenie hlavičky paketu, descriptorov a inputu zariadenia, a ako posledné už vyššie spomínaný Color Map. V našom konkrétnom prípade máme aj 3 odlišné modely – jeden pre každý view. Vzhľadom na to, že sa opäť pohybujeme v Qt model/view frameworku, riešime iba vyobrazenie dát. Keďže nám vyhovuje štandardný spôsob vyobrazenia dát, definujeme len niekoľko základných vlastností pre jednotlivé viewery:

- Každému vieweru nastavíme jemu odpovedajúci model pomocou `setModel()`.

- Nastavíme šírku jednotlivých stĺpciekov buď pomocou *resizeColumnToContents()* alebo *setColumnWidth()*.

Príklad výslednej stromovej štruktúry môžeme vidieť na obrázku 3.13.

Additional Transfer Data		
Data	Meaning	Value
> CONFIGURATION_DESCRIPTOR		
> INTERFACE_DESCRIPTOR		
> HID_DESCRIPTOR		
▼ ENDPOINT_DESCRIPTOR		
07	bLength	7
05	bDescriptorType (ENDPOINT_DESCRIPTOR)	5
> 81	bEndpointAddress	129
> 03	bmAttributes	3
> 04 00	wMaxPacketSize	4
0A	bInterval	10
> INTERFACE_DESCRIPTOR		
> HID_DESCRIPTOR		

Obr. 3.13: Vyobrazenie stromovej štruktúry

4. Vývojová dokumentácia

4.1 Architektúra aplikácie

4.2 Jadro aplikácie

4.2.1 USB_Packet_Analyzer

riadi celkový beh programu, reaguje na input od užívateľa

4.2.2 Item Manager

spracovanie samostatného packetu a uloženie dát o ňom

4.2.3 DataView

trieda ktorá má na starosti vyskakovacie okno po dvojkliku a item a následne reaguje na input od užívateľa v okne

4.2.4 TreeItem

reprezentuje jednotlivé nody v stromovej štruktúre ktorá sa potom využíva na zobrazenie dát v QTreeView

4.3 Modely

4.3.1 AdditionaldataModel

model na spravovanie zvyšných dát (dát ktoré nie sú súčasťou hlavicej packetu)

4.3.2 ColorMapModel

vyobrazenie pomocnej mapy na lepšie sa zorientovanie v zvyraznenom hexdumpe

4.3.3 DataViewModel

model na hexdump - prenáša hex/printable a zároveň o čo vlastne ide (konkrétny descriptor, interrupt data, ...)

4.3.4 TreeItemBaseModel

model na QTreeView ktorým využíva TreeItem

4.3.5 USBPcapHeaderModel

model na QTreeView ale špeciálne pre USBPcap hlavicku packetu

4.4 Interpretery

4.4.1 BaseInterpreter

abstractna trieda od ktorej dedia vsetkz interpretery

4.4.2 Interpreter factory

factory trieda na pridelenie konkretného interpreteru za runtimu kvoli jednoduchosti na lepsie rozsirenie programu do buducnosti

4.4.3 Interpretery descriptorov

Config,Device,Setup,String,...

4.4.4 Interrupt transfer interpretery

obecne interrupt transfer interpreter - sluzi skor ako factory na rozne doteraz implementovane HID zariadenia

Joystick interpreter

Mouse interpreter

Keyboard interpreter

4.5 Delegáti

DataViewerDelegate

Qt delegat - stara sa o highlight hexdumpu

4.6 HID

4.6.1 HIDDevices

staticka trieda, drzi vsetky rozpoznane HID zariadenia a obsahuje funkcie specificke nich - parsovanie HID Report descriptoru

4.7 Práca so súbormi

4.7.1 FileReader

praca zo suborom a predavanie precitanych dat, offline/online capture, QFile vs std::istream

4.8 Globálne dáta

4.8.1 ConstDataHolder

staticka trieda na drzanie si konstant ktore su potrebne napriec celym programom. Mapovanie z enumu do jeho stringovej reprezentacie

4.8.2 PacketExternStructs

obsahuje definiciu vsetkych dolezitych USBPcap structov, pcap structov, enumov a vsetkych structov ktore pouzivam v aplikacii

5. Možnosti rozšírenia

Rozobrať čo všetko sa dá urobiť s tými dátami, ktoré už mám uložené v pamäti, ale momentálne sa s nimi nič nedeje

5.1 Ukladanie výstupu do súboru

výstup analýzy do súboru(textového)

5.2 Iná vizuálna reprezentácia dát

Momentálne vyzobrazujem dáta prevažne v QTreeView alebo QTableView, ale vďaka tomu ako ich mám uložené + to že nad nimi operuje nejaký model ktorý vie vrátiť dáta na základe indexu, by nemuselo byť taká zložitá pridať inú vizualizáciu dát(napríklad obrázkovú ako tu : https://www.usbmadesimple.co.uk/ums_5.htm)

5.3 Pridávanie nových interpreterov pre descriptor

pridanie nových druhov descriptorov - pridať nový interpreter do factory

5.4 Pridanie interpreteru na interrupt transfer

pridanie analýzy interrupt transferu aj pre ine ako hid zariadenia

5.4.1 Pridanie nových HID zariadení

nove HID zariadenie - pridanie do interrupt "factory"

5.5 Pridanie analýzy pre isochronous a bulk transfer

semantická analýza aj iných ako interrupt alebo control transferov - momentálne sú rozpoznávané len v hexdumpe

5.6 ?Možnosť rozšírenia na iné platformy?

uprava aplikácie aby bola prenositeľná aj na iné platformy, čo všetko by tam bolo treba upraviť(pravdepodobne nie veľa, keďže Qt je prenosné, a prakticky jedine čo používam spojené s Windowsom sú jeho struktúry na rôzne descriptor)

6. Užívateľská dokumentácia

6.1 Inštalácia

nastavenie celkovej aplikácie, ale aj nainštalovanie USBPcap + wireshark a ich kombinácia pre live capture

6.2 Orientácia v GUI aplikácie

popis k jednotlivým tlačidlám gui

6.3 Používanie aplikácie

ako spustiť live/offline capture, a celkovo ako pracovať s aplikáciou (popis funkcií - doubleClick na item => zobrazí sa pop-up okno s bližšou analýzou)

7. Záver

7.1 Zhrnutie

celkove zhrnutie prace, ?praca s Qt?

7.2 Budúce plány

Zoznam použitej literatúry

- [1] PS/2 port. https://en.wikipedia.org/wiki/PS/2_port.
- [2] Paralelný port. https://en.wikipedia.org/wiki/Parallel_port.
- [3] RS-232 port. <https://en.wikipedia.org/wiki/RS-232>.
- [4] Paralelná SCSI zbernica. https://en.wikipedia.org/wiki/Parallel_SCSI.
- [5] USB 2.0 Specification. <https://www.usb.org/document-library/usb-20-specification>. [súbor usb_20.pdf, Figure 4-1].
- [6] USB 2.0 Specification – definícia USB Host a s ním súvisiace pojmy . <https://www.usb.org/document-library/usb-20-specification>. [súbor usb_20.pdf, kap. 4.1.1.1].
- [7] USB 2.0 Specification – USB zariadenie definícia. <https://www.usb.org/document-library/usb-20-specification>. [súbor usb_20.pdf, kap. 4.1.1.2].
- [8] USB 2.0 Specification – USB Hub definícia. <https://www.usb.org/document-library/usb-20-specification>. [súbor usb_20.pdf, str. 6].
- [9] USB 2.0 Specification – USB Function definícia. <https://www.usb.org/document-library/usb-20-specification>. [súbor usb_20.pdf, str. 6].
- [10] USB 2.0 Specification – USB topológia zbernice. <https://www.usb.org/document-library/usb-20-specification>. [súbor usb_20.pdf, kap. 4.1.1].
- [11] USB 2.0 Specification – USB descriptor definícia. <https://www.usb.org/document-library/usb-20-specification>. [súbor usb_20.pdf, kap. 9.5].
- [12] Zoznam definovaných USB tried. <https://www.usb.org/defined-class-codes>.
- [13] USB 2.0 Specification – USB paket definícia. <https://www.usb.org/document-library/usb-20-specification>. [súbor usb_20.pdf, str. 7].
- [14] USB 2.0 Specification – USB typy prenosov definícia. <https://www.usb.org/document-library/usb-20-specification>. [súbor usb_20.pdf, kap. 4.7].
- [15] USB Client Driver. <https://docs.microsoft.com/en-us/windows-hardware/drivers/usbcon/usb-driver-development-guide>. [sekcia „Where applicable“].
- [16] USB Request Block. <https://docs.microsoft.com/en-us/windows-hardware/drivers/usbcon/communicating-with-a-usb-device>.

- [17] USB 2.0 Specification. <https://www.usb.org/document-library/usb-20-specification>. [súbor usb_20.pdf].
- [18] USB Human Interface Device Class. <https://www.usb.org/document-library/device-class-definition-hid-111>. [str. 9].
- [19] Genius myš použitá v úvode pri porovnaní existujúcich analyzátorov. <https://us.geniusnet.com/product/dx-120/>.
- [20] Logitech myš použitá na ukážku konkrétneho inputu zariadenia. <https://www.amazon.com/Logitech-Spectrum-Shifting-Personalized-Programmable/dp/B0190B663A>.
- [21] Fotka genius myši prevzatá z oficiálnej genius stránky. https://us.geniusnet.com/wp-content/uploads/sites/2/2020/01/DX-110_P18_980x600.jpg.
- [22] Fotka logitech myši prevzatá zo stránky obchodu. https://images-na.ssl-images-amazon.com/images/I/31qPw4sF6uL._AC_.jpg.
- [23] Komunikácia HID Clienta s HID Class driverom. <https://docs.microsoft.com/en-us/windows-hardware/drivers/hid/opening-hid-collections>.
- [24] HID Application Programming Interface (API). <https://docs.microsoft.com/en-us/windows-hardware/drivers/hid/introduction-to-hid-concepts>.
- [25] Tabuľka HID zariadení a ich Access Mode. <https://docs.microsoft.com/en-us/windows-hardware/drivers/hid/hid-architecture>. [sekcia „HID Clients Supported in Windows“].
- [26] libusb knižnica. <https://libusb.info/>.
- [27] HIDAPI knižnica. <https://github.com/signal11/hidapi>.
- [28] Moufiltr – filter driver. <https://docs.microsoft.com/en-us/samples/microsoft/windows-driver-samples/mouse-input-wdf-filter-driver-moufiltr/>.
- [29] Kbfiltr – filter driver. <https://docs.microsoft.com/en-us/samples/microsoft/windows-driver-samples/keyboard-input-wdf-filter-driver-kbfiltr/>.
- [30] USBPcap sniffer. <https://desowin.org/usbpcap/>.
- [31] Pcap file format. <https://wiki.wireshark.org/Development/LibpcapFileFormat>.
- [32] Report Descriptor. https://www.usb.org/sites/default/files/hid1_11.pdf. [str. 33].

- [33] HID Usage Tables pre USB. https://usb.org/sites/default/files/hut1_2.pdf.
- [34] Konkrétny príklad Report Descriptoru myši. https://www.usb.org/sites/default/files/hid1_11.pdf. [str. 71].
- [35] Dear ImGui. <https://github.com/ocornut/imgui>.
- [36] SFGUI. <https://github.com/Tank0s/SFGUI>.
- [37] Qt. <https://www.qt.io/>.
- [38] Qt Model/View architektúra. <https://doc.qt.io/qt-5/model-view-programming.html>.
- [39] Návrhový vzor Model-View-Controller. <https://en.wikipedia.org/wiki/Model-view-controller>.
- [40] Qt QFileSystemModel class. <https://doc.qt.io/qt-5/qfilesystemmodel.html>.
- [41] Qt QTableView class. <https://doc.qt.io/qt-5/qtableview.html>.
- [42] Qt signal a slot. <https://doc.qt.io/qt-5/signalsandslots.html>.
- [43] Npcap. <https://nmap.org/npcap/>.
- [44] Npcap API. <https://nmap.org/npcap/guide/wpcap/pcap.html>.
- [45] C++ Input file stream class – std::ifstream. <https://www.cplusplus.com/reference/fstream/ifstream/>.
- [46] Qt QFile class. <https://doc.qt.io/qt-5/qfile.html>.
- [47] Qt QByteArray class. <https://doc.qt.io/qt-5/qbytearray.html>.
- [48] Qt QVariant class. <https://doc.qt.io/qt-5/qvariant.html>.
- [49] Qt QListWidget class. <https://doc.qt.io/qt-5/qvariant.html>.
- [50] Qt QListView class. <https://doc.qt.io/qt-5/qlistview.html>.
- [51] Qt QListWidgetItem class. <https://doc.qt.io/qt-5/qlistwidgetitem.html>.
- [52] Qt ItemDataRole enum. <https://doc.qt.io/qt-5/qt.html#ItemDataRole-enum>.
- [53] Qt QTableWidget class. <https://doc.qt.io/qt-5/qtablewidget.html>.
- [54] Qt QTableWidgetItem class. <https://doc.qt.io/qt-5/qtablewidgetitem.html>.
- [55] Qt QAbstractScrollArea class. <https://doc.qt.io/archives/qt-5.11/qabstractscrollarea.html>.

- [56] Qt QTreeView class. <https://doc.qt.io/qt-5/qtreeview.html>.
- [57] Qt QAbstractItemView class. <https://doc.qt.io/qt-5/qabstractitemview.html>.

Zoznam obrázkov

1.1	USB topológia. Obrázok prevzatý z USB 2.0 špecifikácie [5].	4
1.2	Ukážka myši, ktorých input budeme porovnávať	6
1.3	Ukážka hexdumpu so zvýrazneným inputom genius myši.	7
1.4	Ukážka hexdumpu so zvýrazneným inputom logitech myši.	7
1.5	Ukážka hexdumpu so zvýrazneným inputom genius myši s významom.	7
1.6	Ukážka hexdumpu so zvýrazneným inputom logitech myši s významom.	7
1.7	Ukážka hexdumpu vo Wiresharku.	8
1.8	Ukážka hexdumpu so zvýrazneným endpoint descriptorom.	8
1.9	Ukážka hexdumpu s farebným oddelením na základe významu.	9
1.10	Ukážka reprezentácie dát pomocou stromovej štruktúry.	9
1.11	Endpoint descriptor reprezentovaný dátami zvýraznenými na obrázku 1.7 vyššie.	9
1.12	Ukážka vyobrazenia jednotlivých bytov.	10
1.13	Ukážka kliknutia na položku v hexdumpe.	10
1.14	Ukážka kliknutia na položku <i>endpoint descriptoru</i> v stromovej štruktúre.	10
1.15	Príklad obecného vyobrazenia jednotlivých paketov vo Wiresharku.	10
1.16	Príklad inputu myši vo Wiresharku.	11
1.17	Ukážka stromovej štruktúry na zvolenie si zariadenia, s ktorým bude zachytávaná komunikácia.	11
1.18	Príklad hexdumpu v Device Monitoring Studio.	12
1.19	Príklad analýzy paketov.	12
1.20	Ukážka vyobrazenia URB.	12
1.21	Príklad inputu myši v Device Monitoring Studio.	12
1.22	Príklad obecného vyobrazenia jednotlivých paketov v Device Monitoring Studio.	13
1.23	Užívateľské rozhranie Device Monitoring Studio.	13
3.1	Tabuľka zariadenia ich <i>Access Mode</i> . Zariadenia postupne po riadkoch – myš, joystick a klávesnica	19
3.2	Konkrétny príklad Report Descriptoru myši prevzatý z HID Class Specification [34]	22
3.3	Časť Report Descriptoru reprezentujúca jednotlivé kolekcie	22
3.4	Časť Report Descriptoru reprezentujúca tlačidlá myši	23
3.5	Časť Report Descriptoru reprezentujúca osy X a Y pri pohybe s myšou	23
3.6	Časť Report Descriptoru reprezentujúca ukončenie jednotlivých kolekcií	24
3.7	Vyobrazenie základných informácií o pakete pomocou QListWidget	28
3.8	Vyobrazenie základných informácií o pakete pomocou QTableWidgetItem	28
3.9	Vyobrazenie základných informácií o pakete pomocou QTableWidgetItem	30
3.10	Vyobrazenie hexdumpu s farebným oddelením dát podľa významu	31
3.11	Časť Color Map	32

3.12	Vyobrazenie hexdumpu bez farebného oddelenia dát podľa významu	32
3.13	Vyobrazenie stromovej štruktúry	33

Zoznam tabuliek

Seznam použitých zkratek

Prílohy

.1 První příloha