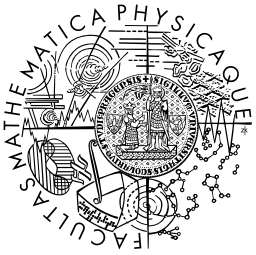


MATEMATICKO-FYZIKÁLNÍ
FAKULTA
Univerzita Karlova

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 zariadnením pripojeným na danú zbernicu a počítačom.

Aplikácia prehľadným spôsobom vizuálne zobrazuje zanalyzované dáta – konkrétne sa 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.

Dôležitá vlastnosť aplikácie je parsovanie HID Report Descriptoru vďaka ktorému bude v budúcnosti 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 Analyzátor

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: The USB bus is the most common way of connecting peripherals to personal computers. The goal of this thesis is to create an application which analyzes communication between a device connected to this bus and a computer.

The application is capable to readably display analyzed data, with specific focus on HID class devices. The application implements semantic analysis of a subset of HID devices consisting of mice, keyboards and joysticks. The methods that the application uses to visually represent data are inspired by already existing applications, where our application combines them and impoves their capabilities to achieve better results.

Notable part of the application is its ability to parse HID Report Descriptor, to accomplish easier addition of new HID devices for semantic analysis. Overall design of the application is general enough to allow simple addition of analysis for other USB classes.

Keywords: USB HID Analyzer

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 **16**
- 2.1 Komunikácia . . . 16
- 2.2 Konfigurácia . . . 17
- 2.3 USB Descriptory . . . 17
- 2.4 Windows . . . 18

3 Analýza **22**
- 3.1 Získanie USB packetov . . . 22
- 3.2 Sémantická analýza dát . . . 24
- 3.3 Voľba frameworku . . . 29
- 3.4 Spracovávanie pcap súborov . . . 30
- 3.5 Uchovávanie informácií . . . 31
- 3.6 Spracovávanie live capture . . . 32
- 3.7 Zobrazenie základných informácií . . . 33
- 3.8 Hexdump . . . 36
- 3.9 Zobrazenie sémantického významu dát . . . 38

4 Vývojová dokumentácia **40**
- 4.1 Kompilácia . . . 40
  - 4.1.1 Inštalácia Qt . . . 40
  - 4.1.2 Visual Studio 2019 a Qt . . . 41
  - 4.1.3 Warningy pri kompilácii . . . 43
- 4.2 Architektúra aplikácie . . . 44
- 4.3 USB_Packet_Analyzer . . . 44
  - 4.3.1 Užívateľská interakcia . . . 45
  - 4.3.2 ItemManager . . . 46
- 4.4 DataViewer . . . 48
  - 4.4.1 Model . . . 49
  - 4.4.2 Delegát . . . 49
  - 4.4.3 Hexdump . . . 50
  - 4.4.4 Hexdump delegát . . . 51
  - 4.4.5 TreeItem . . . 51
  - 4.4.6 TreeItemBaseModel . . . 52
  - 4.4.7 ColorMapModel . . . 53
  - 4.4.8 USBPCapHeaderModel . . . 53
  - 4.4.9 AdditionalDataModel . . . 54
  - 4.4.10 BaseInterpreter . . . 54
  - 4.4.11 InterpretFactory . . . 56
- 4.5 HIDDevices . . . 57

4.6 DataHolder . . . 57
4.7 PacketExterStructs.hpp . . . 58
4.7.1 ExternStructs . . . 58

5 Možnosti rozšírenia **60**
5.1 Ukladanie výstupu do súboru . . . 60
5.2 Iná vizuálna reprezentácia dát . . . 61
5.3 Pridávanie nových Interpreterov pre descriptory . . . 62
5.4 Pridanie Intepreteru pre Interrupt Transfer . . . 63
5.5 Pridanie analýzy pre Isochronous a Bulk Transfer . . . 63
5.6 Možnosť rozšírenia na iné platformy . . . 63

6 Užívateľská dokumentácia **65**
6.1 Inštalácia . . . 65
6.2 Orientácia v GUI aplikácie . . . 65
6.3 Používanie aplikácie . . . 65

7 Záver **66**
7.1 Zhrnutie . . . 66
7.2 Budúce plány . . . 66

Zoznam použitej literatúry **67**

Zoznam obrázkov **72**

Zoznam tabuliek **74**

Seznam použitých zkratek **75**

Prílohy **76**
.1 První příloha . . . 77

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 čiastočné 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 prispôsobený 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, ktoré 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.

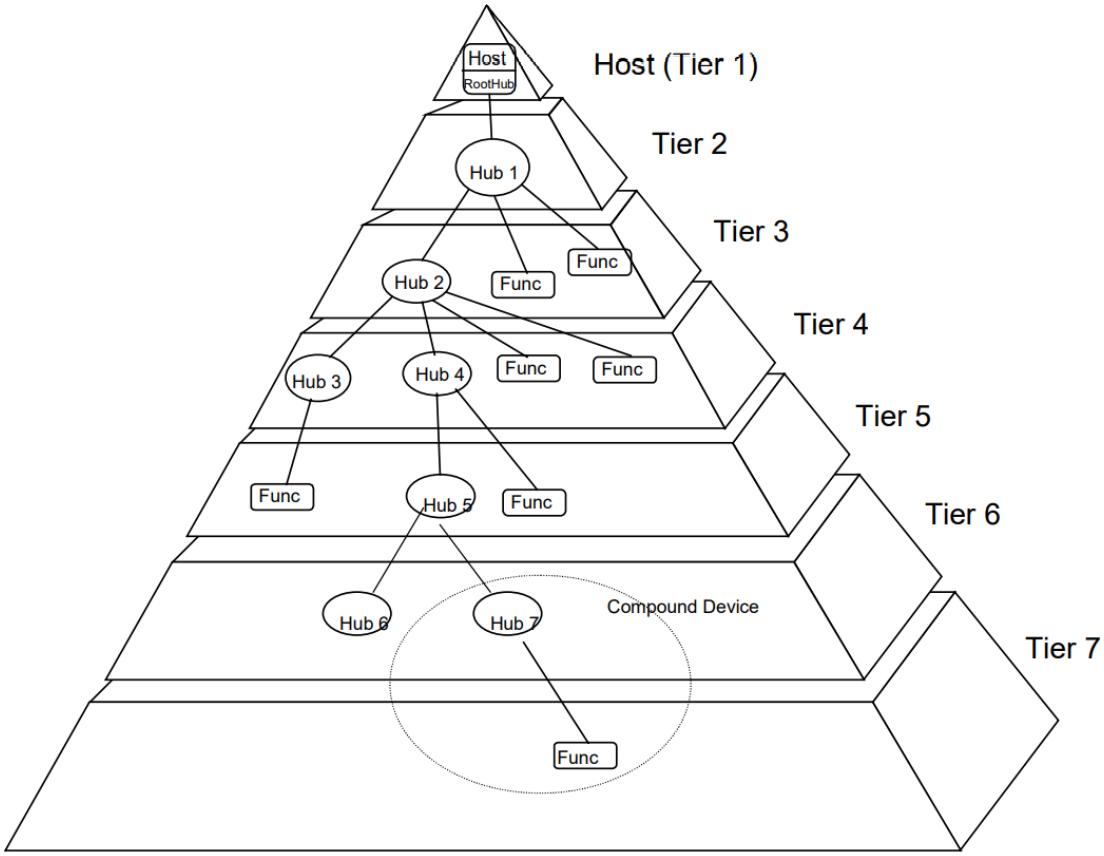
1.1 Základné pojmy

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 funkcionalitu systému (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. **descriptory** 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 funkcionalitu. Tieto triedy umožňujú *USB hostovi* identifikovať dané zariadenie a jeho funkcionalitu. 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 descriptory, 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 konfiguráciu *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 kokrétnu platformu.

Podľa MSDN dokumentácie [15] je **USB client driver** software nainštalovaný na počítači, ktorý komunikuje s USB zariadením aby spojazdnil jeho funkcionalitu. 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 požiadavku, takisto ako aj status o jeho vykonaní.

Na záver si ešte zadefinujeme 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 funkcionalita 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 systémovových aplikácií. Medzi najpoužívanejšie príklady patrí myš, klávesnica alebo joystick.

Ako sme už spomínali vyššie, jednotlivé USB triedy majú definované vlastné descriptory špecifické pre danú triedu. Jedným z takýchto descriptorov je aj *Report Descripor*, ktorý popisuje dáta generované konkrétnym zariadením. Analýzou *Report Descriporu* sme schopní určiť veľkosť a kompozíciu dát posielaných zariadením. Z toho vyplýva, že komunikácia HID zariadenia 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 (obrázok 1.2) – Genius DX-120 [19] a Logitech G502 Proteus Spectrum [20]

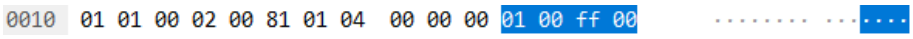
(a) Fotka genius myši prevzatá z oficiálnej genius stránky [21]

(b) Fotka logitech myši prevzatá zo stránky obchodu [22]

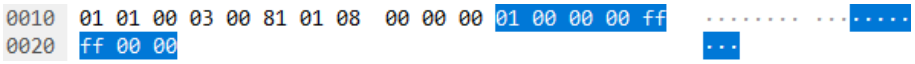
Obr. 1.2: Ukážka myší, ktorých input budeme porovnávať

Teraz si ukážeme ako sa líši ich input. Dáta budeme vyobrazovať pomocou obyčajného hexdumpu (zvýraznená časť v hexdumpe reprezentuje input zariadenia). Na oboch myšiach budeme mať stlačené 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. Už na prvý pohľad môžeme vidieť, že dáta majú rozdielnu dĺžku a obecne o ich význame nevieme nič povedať – ten je definovaný v *Report Descriptore*.



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



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

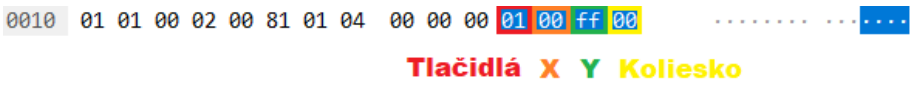
Aj napriek tomu, že sa jedná o zariadenia z tej istej USB triedy a dokonca o rovnaké zariadenie – myš, má ich komunikácia rozličný tvar definovaný priamo výrobcom zariadenia. Analýzou *Report Descriporu* (o ktorej si viac povieme v sekcii 3.2) sme zistili, že dáta, ktoré poslala Genius myš majú nasledujúci 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

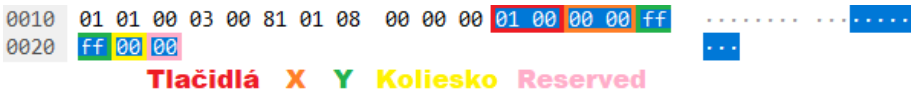
Pre porovnanie, význam dát poslaných Logitech myšou je nasledovný:

- 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.



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



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 Descripor* a na základe toho interpretovať input zariadení.

1.2 Existujúce aplikácie

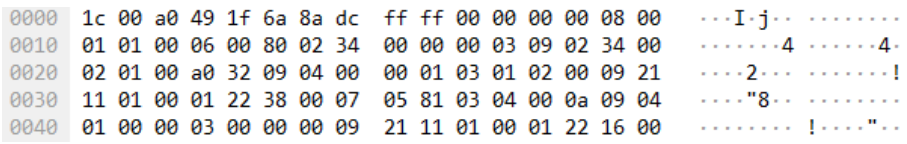
Momentálne existuje niekoľko známych aplikácií ktoré slúžia na analýzu USB paketov. Ich predbežným skúmaním a používaním sme ale zistili, že úplne nevyhovujú našim konkrétnym požiadavkám. Avšak mnohé ich funkcie nám prídu užitočné a môžu poslúžiť ako inšpirácia v implementovaní našej aplikácie. V tejto 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 funkcionalitu 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 funkcionalita 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.

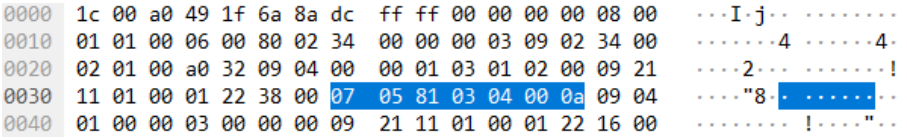
Pre priblíženie niektorých funkcií Wiresharku si ukážeme analýzu komunikácie s USB myšou (Genius DX-120 [19]) [1]. Medzi tie úplne základné funkcie určite patrí hexdump dát nad ktorými prebieha analýza, ktorý je vyobrazený na obrázku 1.7.



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*.

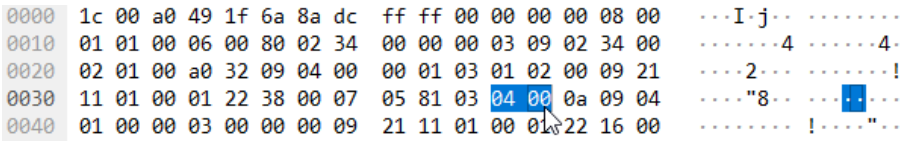


Obr. 1.8: Ukážka hexdumpu so zvýrazneným endpoint descriptorom.

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 časťou bytu

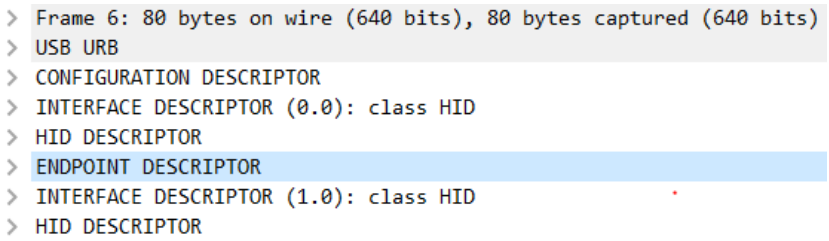
[1] Verzia Wiresharku použítá v tejto sekcii na priblíženie jeho funkcionality je 3.2.6.

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



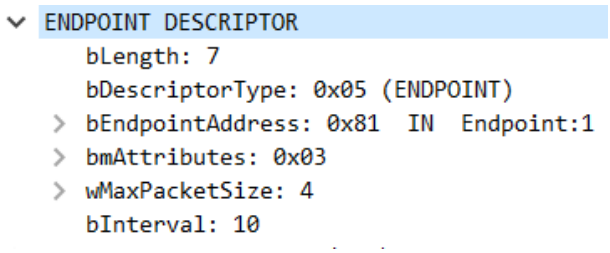
Obr. 1.9: Ukážka hexdumpu s farebným oddelením na základe významu.

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.



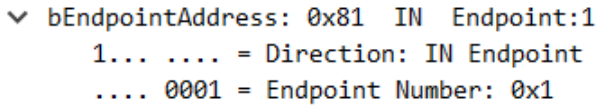
Obr. 1.10: Ukážka reprezentácie dát pomocou stromovej štruktúry.

Jednotlivé položky si môžeme bližšie rozbaliť. 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



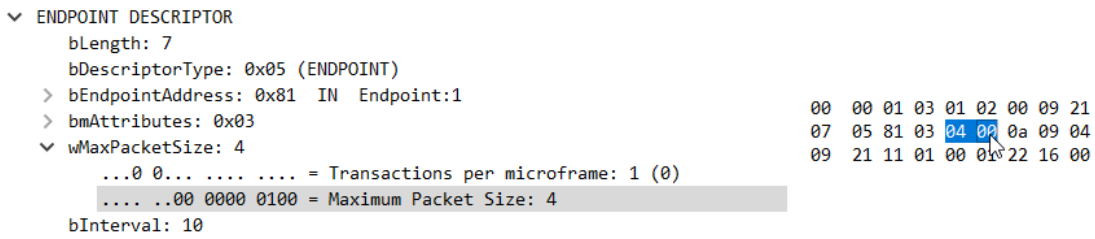
Obr. 1.11: Endpoint descriptor reprezentovaný dátami zvýraznenými na obrázku 1.7 vyššie.

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 $\longrightarrow$ host, OUT opačne) a dolné 4 bity označujú číslo endpointu. Túto vlastnosť aj napriek jej využitiu mnohé konkurenčné aplikácie postrádajú.



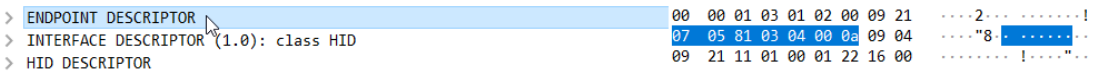
Obr. 1.12: Ukážka vyobrazenia jednotlivých bytov.

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.



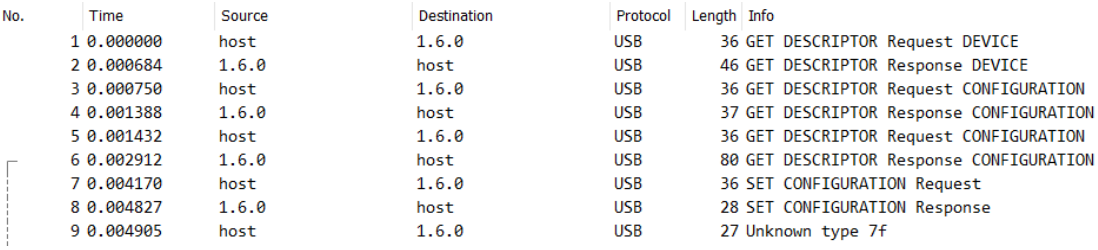
Obr. 1.13: Ukážka kliknutia na položku v hexdumpe.

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.



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.



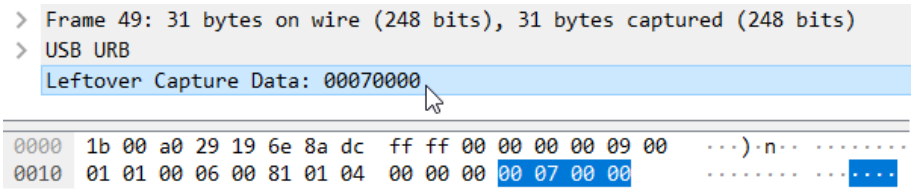
No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	host	1.6.0	USB	36	GET DESCRIPTOR Request DEVICE
2	0.000684	1.6.0	host	USB	46	GET DESCRIPTOR Response DEVICE
3	0.000750	host	1.6.0	USB	36	GET DESCRIPTOR Request CONFIGURATION
4	0.001388	1.6.0	host	USB	37	GET DESCRIPTOR Response CONFIGURATION
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 [2]. Zo sekcie 1.1 nám je teda jasné, že vôbec netušíme čo jednotlivé dáta znamenajú, pretože ich význam je definovaný v *Report Descriptore*.

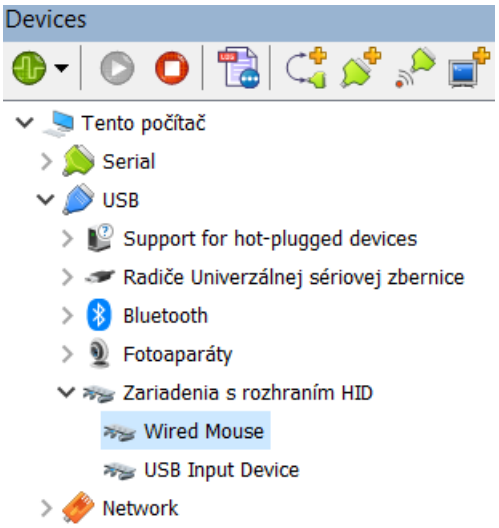


Obr. 1.16: Príklad inputu myši vo Wiresharku.

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 [3].

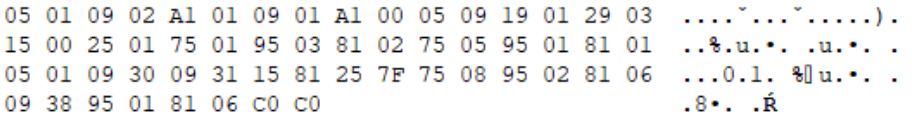


Obr. 1.17: Ukážka stromovej štruktúry na zvolenie si zariadenia, s ktorým bude zachytávaná komunikácia.

[2] V čase dokončovania tejto práce bola k dispozícii verzia Wiresharku 3.4.5, ktorá ponúkala sémantickú analýzu inputu HID zariadení.

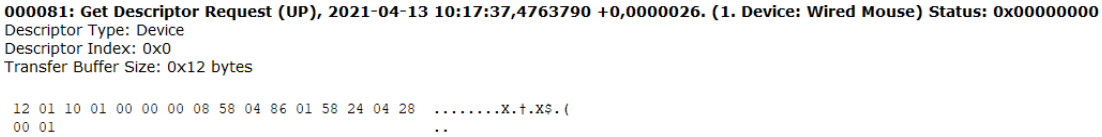
[3] Verzia programu použítá v tejto sekcii na priblíženie jeho funkcionality je 8.36.00.9618

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.



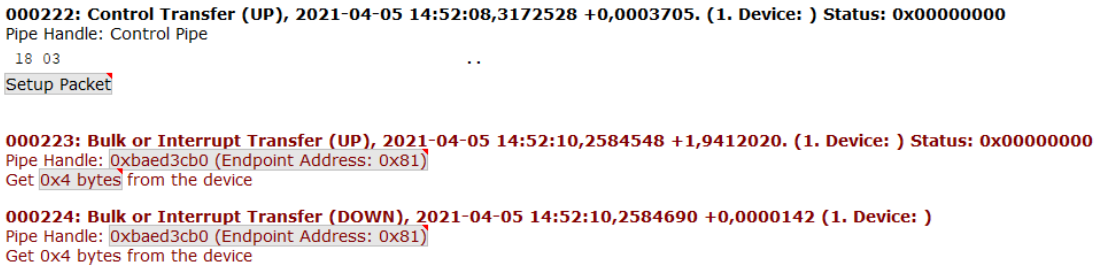
Obr. 1.18: Príklad hexdumpu v Device Monitoring Studio.

Takisto tu nemáme kompletné 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.



Obr. 1.19: Príklad analýzy paketov.

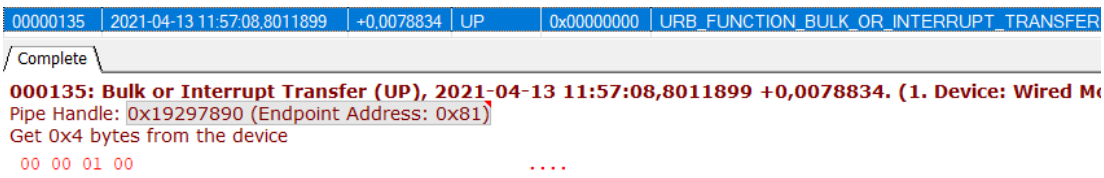
Vyobrazenie *URB* (obrázok 1.20) tak ponúka súhrn týchto popisov jednotlivých paketov, ktoré sú postupne zachytené počas komunikácie na zbernici.



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.



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 funkcionalita, 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 funkcionalita, ktorú ale program ponúka len v platenej verzii, je umožnenie užívateľovi priamo komunikovať so zvoleným zariadením. Môžeme mu tak posielať rôzne požiadavky (niektoré z nich sú spomenuté v USB 2.0 špecfikácii[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 zadefinovať platformu na ktorú budeme cieliť 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 zobraziť 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 naledujúce 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 zadefinujeme 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 descriptory 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 Studia, kde boli jednotlivé pakety farebne rozlišiteľné, čo zvyšovalo celkový prehľad pohybu paketov na zbernici. Z toho dostávame nasledujúce požiadavky:

P10 Mala by na prvý pohľad jasne zobraziť základné informácie o každom analyzovanom pakete (ako napr. dĺžka paketu, typ prenosu a pod.) a pri bližšom skúmaní jednotlivých paketov detailnejšie zobraziť 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 sématnicky reprezentovať input nami zvolených HID zariadení – myš, klávesnica a joystick.

1.4 Ciele práce

Celkové ciele tejto práce sú následovné :

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

V tejto kapitole si rozšírime znalosti fungovania USB, ktoré sme získali v sekcii 1.1. Najprv si vysvetlíme ako prebieha komunikácia medzi USB zariadením a hostom, následne sa pozrieme na konfiguráciu USB zariadenia po pripojení na zbernicu, prejdeme si niektoré základné USB descriptory a ako posledné si ešte vysvetlíme základnú stavbu USB na Windowse.

2.1 Komunikácia

Detailný popis komunikácie a presunu dát po zbernici je opísaný v USB 2.0 špecifikácii v kapitole 5 [23]. My sa momentálne zameriame len na určité časti, ktoré sú potrebné pre našu prácu. Každé USB zariadenie v sebe obsahuje tzv. *endpointy* [24] – môžeme to považovať za akúsi koncovku komunikácie medzi hostom a zariadením. Koncepčne sa jedná o schopnosť zariadenia v sebe uchovať dáta (memory buffer). Ako už vieme z kapitoly 1.1, komunikáciu riadi USB host, a tá prebieha práve pomocou týchto endpointov – v prípade ak chce host poslať určité dáta zariadeniu, zapíše ich do jeho konkrétneho endpointu. V prípade ak chce zariadenie poslať určité dáta hostovi, zapíše si ich do daného endpointu, odkiaľ ich host potom prečíta.

Endpoint

USB zariadenie typicky pozostáva z niekoľkých na sebe nezávislých endpointov. Každý endpoint je potom jednoznačne určený:

1. Adresou USB zariadenia – tá je pridelená USB zariadeniu pri jeho konfigurácii v momente pripojenia na zbernicu.

2. Číslom endpointu – unikátne číslo, ktoré určuje výrobca zariadenia.

3. Smerom prenosu dát – host $\longrightarrow$ device alebo device $\longrightarrow$ host.

Do momentu pokiaľ neprebehne konfigurácia USB zariadenia a jeho endpointov, sú endpointy s adresou inou ako 0 v neurčitom stave a nemusia byť pre hosta dostupné. Endpoint s adresou 0, inak nazývaný aj ako „default endpoint“ alebo „Endpoint0“, slúži na nakonfiguranie daného USB zariadenia. Výrobca zariadenia je povinný poskytnúť aspoň 1 Endpoint0 pre každý smer pohybu dát, prípadne 1 Endpoint0 s možnosťou prenosu oboma smermi. Funkcie (USB zariadenia) môžu obsahovať aj ďalšie endpointy s adresou inou ako 0. Tieto endpointy slúžia na prenos dát špecifických pre dané zariadenie (napríklad na posielane inputu myši). Rôzne endpointy môžeme zlučovať do určitých množín podľa ich funkcionality. Takúto množinu endpointov potom nazývame *interface*. Niektoré USB zariadenia potom môžu pozostávať z viacerých interfacov, ktoré budú reprezentovať rozličné USB triedy.

Pipe

USB pipe [25] je termín označujúci spojenie medzi konkrétnym endpointom USB zariadenia a Host Controllerom (interface medzi hostom a zbernicou). Reprezentuje schopnosť prenášať dáta medzi hostom a endpointom pomocou memory bufferu. Pipe ktorá pozostáva z dvoch Endpoint0 sa nazýva „Default Control Pipe“ a je prístupná v momente pripojenia zariadenia na zbernicu a slúži na konfiguráciu daného zariadenia (po konfigurácii môže mať aj iné využitie, ktoré špecifikuje samotný výrobca). Ostatné pipy s ďalšími endpointami (za predpokladu, že existujú) sú vytvorené až po konfigurácii zariadenia.

2.2 Konfigurácia

Konfigurácia USB zariadenia je detailne opísaná v USB 2.0 špecifikácii [26]. Predtým ako začneme využívať funkcionalitu pripojeného USB zariadenia, je USB host zodpovedný za jeho nakonfigurovanie. Počas konfigurácie posiela USB host zariadeniu tzv. „Device Requesty“, na ktoré dané zariadenie odpovedá cez Default Control Pipe. Tieto requesty sú špecifikované v *Setup Pakete* – štruktúra veľká 8 bytov so štandardným formátom definovaným v USB špecifikácii 2.0 [27]. Existuje niekoľko základných requestov [28], na ktoré musí každe USB zariadenie vedieť reagovať. Patria medzi ne napríklad:

- Get Descriptor – vypýta si od zariadenia konkrétny descriptor, ktorý mu zariadenie pošle ako odpoveď na tento request (za predpokaldu, že daný descriptor existuje)
- Set Configuration – nastaví konkrétnu konfiguráciu zariadeniu

Bežný postup konfigurácie je, že si USB host vypýta rôzne descriptory od zariadenia, ktoré určujú jeho schopnosti (napr. *Configuration Descriptor*, *Interface Descriptor*, *Endpoint Desriptor*) a potom pomocou requestu Set Configuration nastaví požadovanú konfiguráciu (a ak je to nutné, zvolí rôzne dodatočné nastavenie interfacov).

2.3 USB Descriptory

Teraz si prejdeme niekoľko základných USB descriptorov a ich jednotlivé položky, pretože ich význam budeme potrebovať neskôr v tejto práci.

Configuration Descriptor

Configuration Descriptor [29] opisuje informácie o konkrétnej konfigurácii USB zariadenia. Obsahuje položku *bConfigurationValue* – číslo reprezentujúce konkrétnu konfiguráciu, ktorú USB host použije ako parameter
v `SetConfiguration()` requeste v prípade, že chce nastaviť práve túto konfiguráciu. Každé zariadenie má aspoň jeden Configuration Descriptor. Každá konfigurácia obsahuje aspoň jeden interface a každý interface má nula alebo viac endpointov. V prípade, že si host vyžiada od zariadenia Configuration Descriptor, dostane spolu s ním aj všetky súvisiace Interface a Endpoint descriptory.

Interface Descriptor

Interface Descriptor [30] opisuje špecifický interface konkrétnej konfigurácie USB zariadenia. Ak zariadenie podporuje viac ako jeden interface, tak všetky Interface Descriptory spolu s im odpovedajúcimi Endpoint Descriptormi sú vrátené ako odpoveď na `GetConfiguration()` request (k Interface Descriptoru nie je možný priamy prístup pomocou `GetDescriptor()` alebo `SetDescriptor()` requestom). Ak interface používa len Endpoint0, tak za Interface Descriptorom nenasleduje žiaden Endpoint Descriptor. Endpoint0 takisto nie je započítaný v položke Interface Descriptoru *bNumEndpoints*, ktorá udáva počet endpointov konkrétneho interfacu.

Endpoint Descriptor

Endpoint Descriptor [31] poskytuje hostovi informácie o konkrétnom endpointe. Tento descriptor obsahuje aj informácie na základe ktorých je host schopný určiť bandwidth konkrétneho endpointu – množstvo dát prenesených za jednotku času (typicky bity za sekundu = b/s, alebo byty za sekundu = B/s). Takisto ako aj pri Interface Descriptore, nie je možné k nemu priamo pristupovať pomocou `GetDescriptor()` alebo `SetDescriptor()` requestov, ale je súčasťou odpovede na `GetConfiguration()` request.

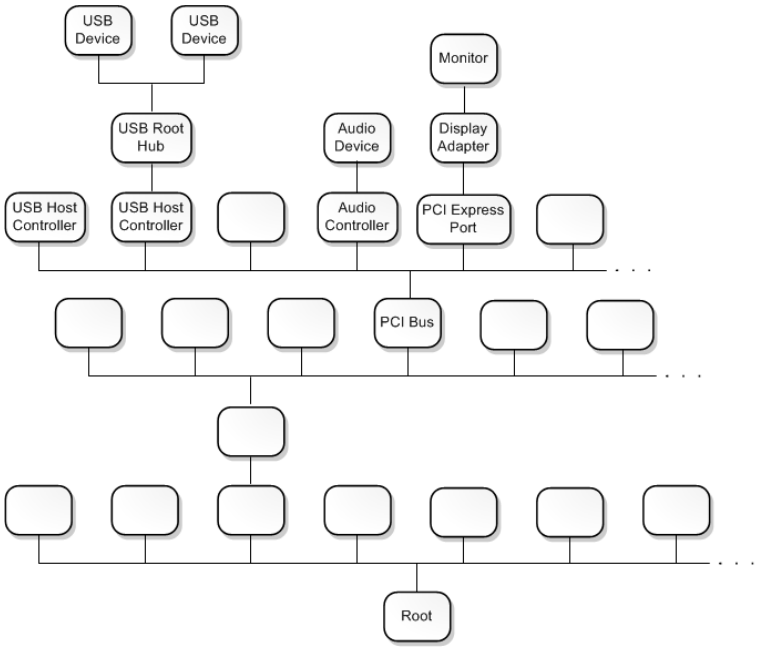
2.4 Windows

Keďže Windows je hlavná platforma na ktorú mierime s našou aplikáciou, priblížime si ako sú reprezentované jednotlivé USB zariadenia a priebeh komunikácie na danej zbernici.

Nasledujúca sekcia poskytuje zjednodušený popis a čerpá (pokiaľ nie je uvedené inak) z toho čo sa nachádza v Microsoft dokumentácii [32]. Windows organizuje zariadenia pomocou stromovej štruktúry nazývanej „Plug and Play device tree“ alebo jednoducho len „device tree“. Správca stromu sa nazýva „PnP manager“ [33]. Vrchol v tomto strome (tzv. „device node“) reprezentuje USB zariadenie alebo nejakú jeho konkrétnu Funkciu. Koreň stromu obecne nazývame „root device node“ a typicky sa v diagramoch kreslí na spodku. Príklad takéhoto diagramu môžeme vidieť na obrázku 2.1 nižšie.

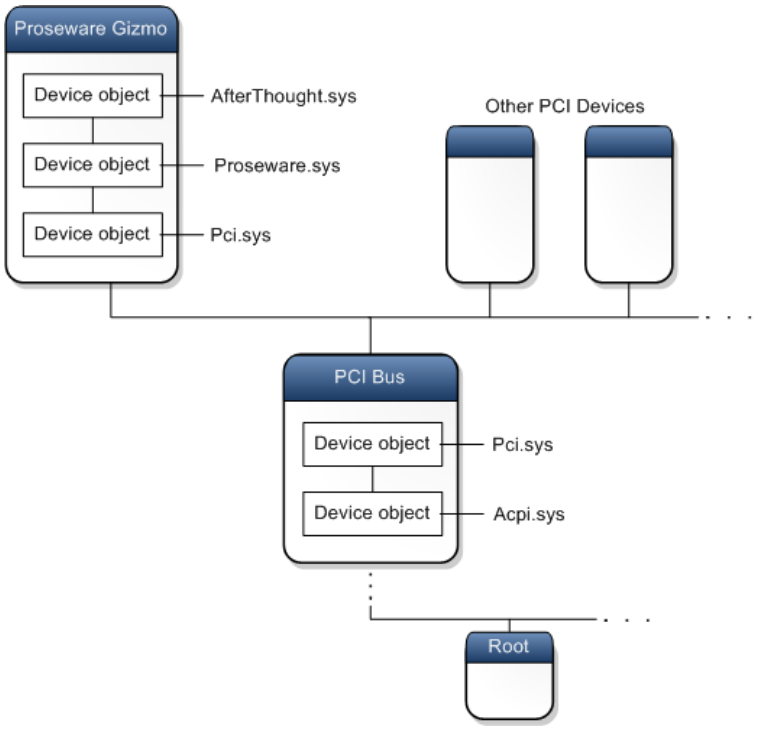
Niektoré vrcholy môžu reprezentovať zbernice samotné, pričom synovia daného vrcholu zase reprezentujú zariadenia pripojené na túto zbernicu.

Windows má definovanú `DEVICE_OBJECT` štruktúru [34] reprezentujúcu *device object* – logické, virtuálne alebo fyzické zariadenie, ktoré využíva driver na spracovávanie I/O (input/output) requestov. Každý device node obsahuje usporiadaný list takýchto device objectov. Usporiadaný list device objectov spolu s ich asociovanými drivermi nazývame *device stack* (môžeme si to predstaviť ako stack dvojíc *device object*↔driver). Tento stack má podľa konvencie vrch aj spodok – prvé zariadenie ktoré bolo vytvorené na stacku sa nachádza na spodku a posledné vytvorené je naopak na vrchu. Ukážeme si to na konkrétnom príklade, kde na nasledujúcom obrázku 2.2 môžeme vidieť device node s názvom „Proseware Gizmo“, ktorého device stack obsahuje 3 položky: Vrchný device object má asociovaný „AfterThought.sys“ driver, stredný device object je spojený s driverom



Obr. 2.1: Ukážka device tree. Obrázok prevzatý z Microsoft dokumentácie [32]

„Proseware.sys“ a posledný má zase „Pci.sys“ driver. Device stack „PCI Bus“ device nodu obsahuje len 2 položky: device node s „Pci.sys“ driverom a device node s „Acpi.sys“ driverom.



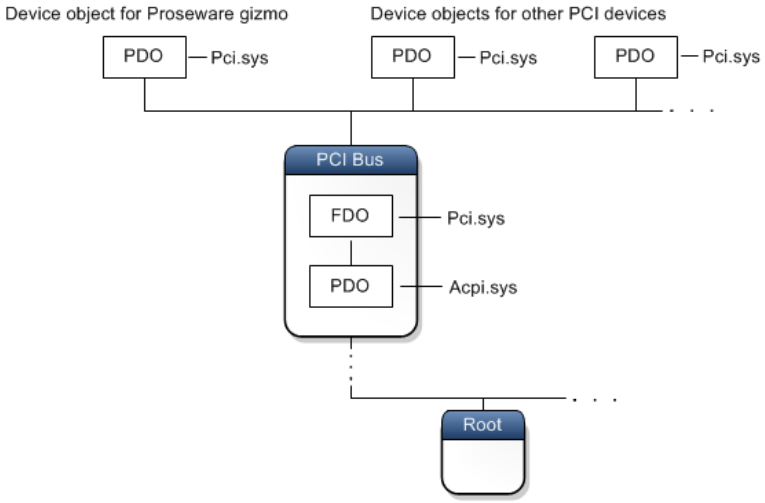
Obr. 2.2: Ukážka konkrétnych device nodov. Obrázok prevzatý z Microsoft dokumentácie [32]

Konštrukcia tohto stacku je nasledovná:

- PnP manager poverí driver každej zbernice (v našom konkrétnom príklade

z obrázku 2.2 by to bol Pci.sys driver) aby sčítala všetky zariadenia, ktoré sú na ňu pripojené.

- Ako odpoveď na túto požiadavku vytvorí driver danej zbernice device object pre každé pripojené zariadenie, ktorý nazývame „physical device object (PDO)“. V príklade s Pci.sys driverom by to vyzeralo nasledovne:



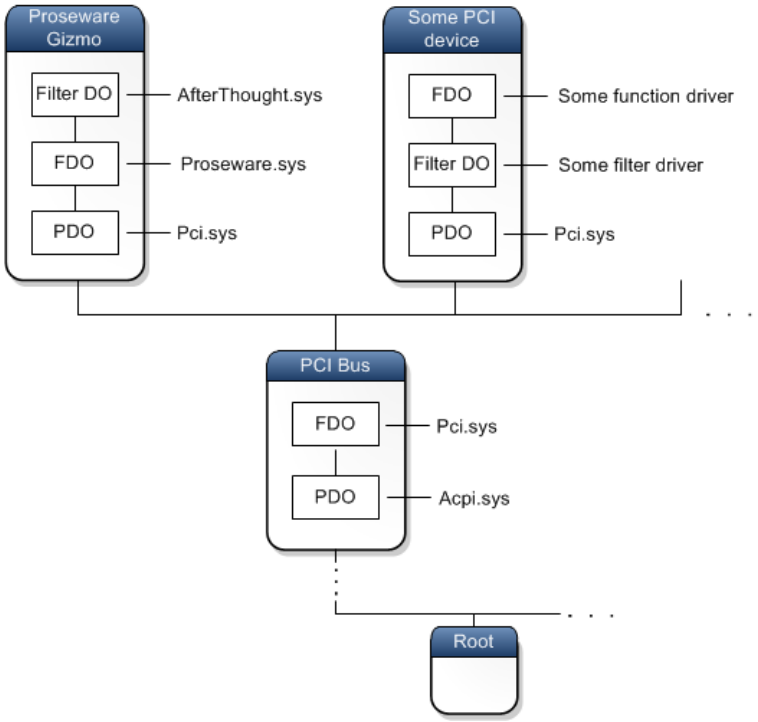
Obr. 2.3: Ukážka vytvorenia konkrétnych PDO Pci.sys driverom. Obrázok prevzatý z Microsoft dokumentácie [32]

- PnP manager pripradí device node každému novo vytvorenému PDO a pozrie sa do registru, aby zistil aké drivery majú byť súčasťou device stacku daného nodu. Každý device stack musí obsahovať práve jeden *function driver* (hlavný driver device stacku, je zodpovedný za riadenie read, write a control requestov) a nula alebo viac *filter driverov* (Ponúkajú prídavné možnosti v spracovávaní read, write a control requestov. Napríklad môžu meniť dáta, ktoré sú posielané daným requestom).

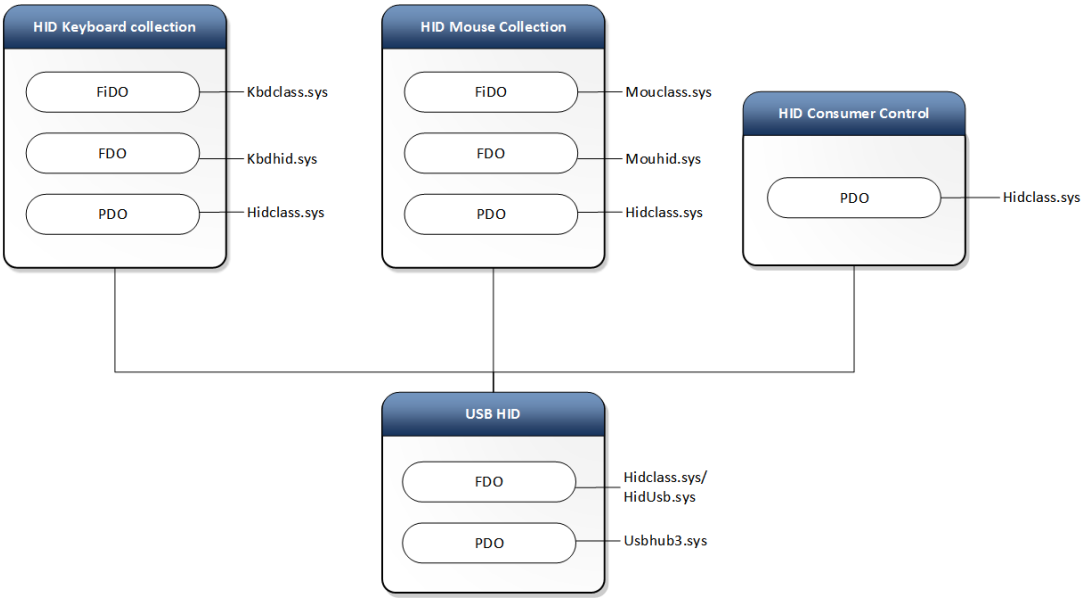
- Akonáhle sú všetky drivery načítané, každý z nich vytvorí odpovedajúci device object a pripojí ho na daný device stack. Device object vytvorený function driverom nazývame „functional device object (FDO)“ a device objet vytvorený filter driverom zase „filter device object (Filter DO)“. Náš konkrétny device tree vyobrazený na obrázku 2.4 by teda vyzeral ako na obrázku 2.4.

Ako is ešte môžeme všimnúť na obrázku 2.4, v device node Proseware Gizmo je fitler driver (AfterThought.sys) nad function driverom (Proseware.sys). V takomto prípade je daný filter driver nazývaný „upper filter driver“. Naopak v druhom device node (Some PCI device) je filter driver pod function driverom – takýto filter driver nazývame „lower filter driver“.

Keďže sú pre našu aplikáciu dôležité hlavne HID zariadenia, pozrieme sa teraz na to ako vyzerá driver stack a celková architektúra pre túto triedu zariadení . Základom každého device stacku HID zariadenia je class driver *hidclass.sys*, za ktorým nasledujú už konkrétne function/filter drivery. Príklad device stacku myši a klávesnice, tak ako aj HID triedy môžeme vidieť na nasledujúcom obrázku 2.5:



Obr. 2.4: Ukážka konkrétneho device tree pre PCI zbernicu. Obrázok prevzatý z Microsoft dokumentácie [32]



Obr. 2.5: Ukážka konkrétneho device stacku myši a klávesnice. Obrázok prevzatý z Microsoft dokumentácie [35]

Momentálne si vysvetlíme ešte zopár pojmov, ktoré sa nám budú hodiť neskôr v texte. „HID Client“ [36] je termín ktorým označujeme driver, service [37] alebo aplikáciu, ktorá komunikuje s *hidclass.sys* a často reprezentuje konkrétne zariadenie (napríklad myš alebo klávesnicu). „Preparsed Data“ [38] reprezentujú dáta Report Descriptoru asociované s konkrétnym zariadením. Aplikácie ich využívajú aby z nich vytiahli informácie o danom zariadení bez nutnosti parsovania Report Descriptoru.

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 obecne 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 paketov nám bude obecne 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 cielime s našou aplikáciou a ktorú sme si zadefinovali v požiadavke **P1**– Windows.

V minulej kapitole 2.4 sme opisovali stavbu HID zariadení vo Windowse, čím sme sa dozvedeli, že device stack každého z nich je postavený na *hidclass.sys*. Microsoft dokumentácia podrobnejšie opisuje komunikáciu medzi HID zaridením a kernel/user-mode aplikáciou [39] za pomoci API tohoto driveru – *HID API* [40]. 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* 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* 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 restrikciu prístupu *HID Clienta* k HID zariadeniu. Ten môže byť buď *Shared* alebo *Exclusive*. *Exclusive Mode* zabraňuje ostatným *HID Clientom* 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* [40] metód `HidD_GetXxx`. Tieto metódy nám obecne umožnia získať niektoré descriptory zariadenia, tak ako aj jeho *Preparsed Data*. Nie je nám ale umožnené volať metódu `ReadFile`, takže nemáme akým spôsobom zachytávať komunikáciu HID zariadenia s clientom.

Tabuľka zariadení [41] (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 zariadenía ich *Access Mode*. Zariadenia postupne po riadkoch – myš, joystick a klávesnica

Známe knižnice

Keďže komunikáciu nemôžeme zachytávať priamo pomocou *HID API* [40] metód, rozhodli sme sa skúsiť použiť niektoré známe knižnice na sledovanie USB zbernice.

Pravdepodobne najznámejšou z nich je *libusb* [42] – knižnica napísaná v jazyku C, ktorá slúži na poskytnutie prístupu k USB zariadeniu. Je cielená 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 privilégiá 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,interrrupt,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 – to nás zaviedlo k *HIDAPI* [43]. *HIDAPI* je ďalšia veľmi rozšírená knižnica, ktorá umožňuje komunikáciu s USB zariadeniami z HID triedy a takisto beží na viacerých patformách 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.

Keďže sme nevedeli získať input zariadenia pomocou *hidclass.sys* driveru a ani pomocu iných third-party knižníc, skúsime sa pozrieť či nemáme k dispozícii filter driver, ktorý by nám poskytol prístup k inputu daného zariadenia.

Filter Driver

Windows ponúka niekoľko vstavaných driverov pre rôzne typy zariadení. Konkrétne pre myš a klávesnicu Windows ponúka 2 filter drivery – *Moufiltr* [44] (upper-level filter driver pre myš) a *Kbfiltr* [45] (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. Ako sme už ale spomínali v úvode, živé zachytávanie paketov má v našej práci najmenšiu prioritu. Preto sme sa rozhodli, že napísanie vlastného USB snifferu je nad rámec tejto práce a použijeme už existujúcu third-party aplikáciu.

Third-party aplikácie

Na zachytávanie USB paketov sme sa rozhodli použiť *USBPcap* [46], 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 obecne známom formáte pcap [47].

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 konkrétneho zariadenia. Zároveň sme si v požiadavke **P12** definovali, že chceme byť schopní rozparsovať HID Report Descriptor pre neskoršiu interpretáciu inputu. 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 [48]. 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 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 [49]. To znamená, že Usage s hodnotou 0x00 môže mať iný význam v Generic Desktop Page ako v Keyboard Page.
- Report Size a Report Count – definujú veľkosť dát Input/Output/Feature itemov. 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 [50], 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 [50]

Z prvej časti (obrázok 3.3) si môžeme všimnúť, že sa celý Report Descriptor delí do logických kolekcií – najprv *Application* a potom *Physical*. Hneď na začiatku vidíme definovanú *Usage Page* s hodnotou *Generic Desktop*, ktorá bližšie špecifikuje nasledujúci *Usage (Mouse)* item – takže už vieme, 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 Page (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 sa prepne *Usage Page* z *Buttons* na *Generic Desktop* a definujú sa 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, čím končí aj celý Report Descriptor.

```
    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 posielať 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 posiela z viacerých endpointov. To si vysvetlíme v nasledujúcej sekcii.

Report Descriptor $\longleftrightarrow$ endpoint

Ako už vieme z minulej kapitoly 2.1, 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 sa zapisujú, resp. z ktorého endpointu sa čítajú 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 posiela 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 pri konfigurácii daného zariadenia, keď si USB host vypýtal od zariadenia Configuration Descriptor a spolu s ním dostal aj Interface Descriptor a jemu odpovedajúce Endpoint Descriptory.

3.3 Voľba frameworku

Pre jazyk C++ existuje hneď niekoľko známych GUI frameworkov nad ktorými sme uvažovali – *Dear ImGui* [51], *SFGui* [52] a *Qt* [53]. Autor tohto projektu nemal so žiadnym z týchto frameworkov predchádzajúcu skúsenosť, ale na základe rešeršu si nakoniec zvolil Qt. Teraz si prejdeme zopár požiadaviek na základe ktorých sme porovnávali jednotlivé frameworky.

Dostupnosť

Jednou zo základných požiadaviek na daný framework je aby bol dostupný úplne zadarmo. Veľakrát môže byť užitočné nahliadnuť do implementácie konkrétnych komponent a metód pre lepšie pochopenie ich fungovania. Práve preto je naša ďalšia požiadavka, aby bol daný framework open-source. Vzhľadom na zameranie našej aplikácie na Windows je pre dôležité aby daný framework fungoval na tejto platforme. Neskôr by sme ale mohli rozmýšľať o rozšírení aj na ďalšie plaformy, a preto by sme si už od začiatku chceli zvoliť framework, ktorý podporuje aj iné platformy, ako napríklad Linux a macOS. Z vyššie spomínaných frameworkov to spĺňajú všetky.

Dokumentácia

Kvalitná dokumentácie je veľmi dôležitá na pochopenie daného frameworku. Tu je Qt jednoznačne lepšie ako ostatné frameworky. Zatiaľ čo ony ponúkajú strohú dokumentáciu na github v podobe dokumentačných komentárov v kóde, Qt má samostatnú stránku na ktorej je podrobne opísaná každá jedna jeho komponenta.

Príklady

Vzhľadom na to, že autor nemal so žiadnym zo zvolených frameworkov predchádzajúcu skúsenosť, sú ilustračné príklady použitia a ich implementácia dobrý spôsob akým začať prácu s frameworkom a zistiť jeho možnosti. Qt tu má zase obrovskú výhodu pretože poskytuje desiatky rozličných príkladov použitia (so širokou škálou obtiažnosti), ktoré sú detailne zdokumentované. Ostatné frameworky poskytujú zopár jednoduchých príkladov uložených na svojom github, ktorých dokumentácia sú iba komentáre v kóde.

Rozsiahlosť

Na veľkosť frameworku sa môžeme pozerať dvoma rozličnými spôsobmi. Na jednu stranu ak je framework príliš rozsiahly, bude ťažšie ho pochopiť a naučiť sa používať. Qt je z daných frameworkov určite najrozsiahlejší. Na druhú stranu poskytuje o to väčšiu funkcionalitu a nemali sme tak problém implementovať

čokoľvek sme chceli. Zároveň poskytuje rôzne ďalšie komponenty, ktoré nesúvisia priamo s implmentáciou grafického rozhrania ale ako uvidíme neskôr, veľmi nám počas vytvárania našej aplikácie pomohli.

Vzhľadom na to, že sme si zvolil Qt musíme spomenúť ešte jednu vec, ktorá je pre tento framework špecifická, a to spôsob akým sa v ňom programuje. Využíva sa tu tzv. model/view architektúra [54].

Model/View Architektúra

Qt obsahuje množinu itemov, ktoré používaju model/view architektúru na riadenie vzťahu medzi reprezentáciou dát a spôsobom akým sú vyobrazené. Celá architektúra vychádza zo známeho návrhového vzoru MVC [55] (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) použijeme na to tzv. *delegate*. Architektúra sa teda dá rozdeliť do 3 hlavných komponent – model, view a delegate. Každá z týchto komponent je definovaná abstraktnou triedou, ktoré ponúkajú jednotný interface a v niektorých prípadoch (napríklad QFileSystemModel [56], QTableView [57], atď.) máme dokonca už predimplementovanú aj základnú funckionalitu. Komunikácia medzi komponentami prebieha pomocou tzv. *signals* a *slots* [58]. Signal je vyvolaný konkrétnym eventom (napríklad stlačenie tlačidla na konrétny item) a slot je funkcia, ktorá sa zavolá ako odpoveď na daný signal.

3.4 Spracovávanie pcap súborov

V sekcii 3.1 sme sa rozhodli, že na zachytávanie paketov budeme používať third-party aplikáciu USBPcap. Tá, ako sme už spomínali, zachytáva pakety do súboru vo formáte pcap, čo je obecne známy formát so špecifickým zameraním práve na ukladanie dát reprezentujúcich pakety. Teraz sa pozrieme na niekoľko možných spôsobov ako daný súbor spracovať:

Npcap API

Npcap [59] je Windows knižnica na zachytávanie sieťových paketov, ktorá poskytuje Npcap API [60] pomocou ktorého 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 dodatočných dll súborov a celkovú integráciu nášho projektu s knižnicou. 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 cielime, 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` [61], keďže ale používame Qt framework, máme k dispozícii triedu *QFile* [62]. QFile ponúka jednoduché API vďaka ktorému môžeme postupne spracovávať ľubovoľný 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 určitým spôsobom zareprezentovať 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 zistíme až za run-timu, 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 s 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* [63]. QByteArray je trieda, 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 ukladaní dát sa budeme bližšie baviť v sekcii 3.7 nižšie ale aby sme to pochopili, vysvetlíme si tu ešte jeden koncept Qt, ktorý neskôr využijeme spolu s QByteArray pri ukladaní dát – *QVariant* [64]. 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 Spracovávanie live capture

Ako sme už spomínali vyššie v sekcii **??**, súbory spracovávame pomocou QFile. Ak máme fixný súbor, ktorý sa počas spracovávania a ani po ňom nemení, tak jeho spracovanie je priamočiare – prechádzame `while` cyklom cez súbor až pokiaľ sa nedostaneme na jeho koniec a tým celé spracovanie končí. Počas live capturu to je o trochu komplikovanejšie, pretože potrebujeme neustále sledovať spracovávaný súbor na aktuálne zmeny. To môžeme riešiť viacerými spôsobmi:

Nekonečný cyklus

Jedno z riešení by bolo celý proces spracovávania súboru zabaliť do nekonečného `while(true)` cyklu. Toto riešenie nie je práve ideálne z dvoch hlavných dôvodov:

1. Záťaž procesora – mohli by sme do kódu pridať napríklad `Sleep(50)`, ale stále to tvorí obrovský nátlak na procesor.

2. User-experience – z užívateľského pohľadu je toto riešenie veľmi nepraktické. Vzhľadom na to, že používame Qt, náš program reaguje na interakciu užívateľa pomocou tzv. „signálov a slotov“ [58]. Zároveň na daný cyklus nevyužívame žiadne metódy paralelného programovania a teda Qt komponenty nemajú možnosť tieto signály emitnúť. Riešenie by bolo volať priebežne metódu `QCoreApplication::processEvents()`, ktorou by sme spracovali dané signály. Takýto prístup má za následok veľmi zlú odozvu vzhľadiska užívateľskej interakcie s aplikáciou, čo pôsobí sekavým dojmom.

QFileSystemWatcher

QFileSystemWatcher [65] Qt trieda ponúka interface na monitorovanie jednotlivých súborov a priečinkov. Pomocou `addPath(QString)` metódy si pridáme cestu k súboru, ktorý chceme sledovať a pokiaľ je daný súbor akýmsi spôsobom

modifikovaný, QFileSystemWatcher emitne signál `fileChanged(QString)`. Museli by sme si teda spojiť tento signál s našim konkrétnym slotom (takže ak bude `fileChanged(QString)` signál emitnutý, zavolá sa naša slot funkcia), v ktorom by sme implementovali čítanie súboru po jeho prípadných zmenách. Toto riešenie sme bohužial nemohli implementovať z dôvodu, akým vo Wiresharku funguje zapisovanie do súboru počas zachytávania USB paketov. Zapisovanie prebieha až v momente keď máme Wireshark otvorený ako hlavné okno, a nie keď len beží na pozadí. To znamená, že by sme museli prepínať medzi našou aplikáciou a Wiresharkom aby bol súbor, do ktorého sa zapisuje modifikovaný a tým pádom by bol emitnutý signál `fileChanged(QString)`.

Qtimer

Riešenie nám poskytne trieda QTimer [66] a jej signál `timeout()`. Ten sa emitne v pravidelných intervaloch určených metódou
`start(std::chrono::miliseconds)` (čím sa zároveň aj spustí daný timer). K tomuto signálu si pripojíme slot, ktorý skontroluje či nebol daný súbor modifikovaný a prípadne spracuje jeho zmeny. Z vyššie opísaných dôvodov pre ktoré sme nemohli použiť QFileSystemWatcher, je QTimer prijateľná alternatíva, ktorá netvorí záťaž procesora a neprejavuje sa ani čisto z užívateľského pohľadu.

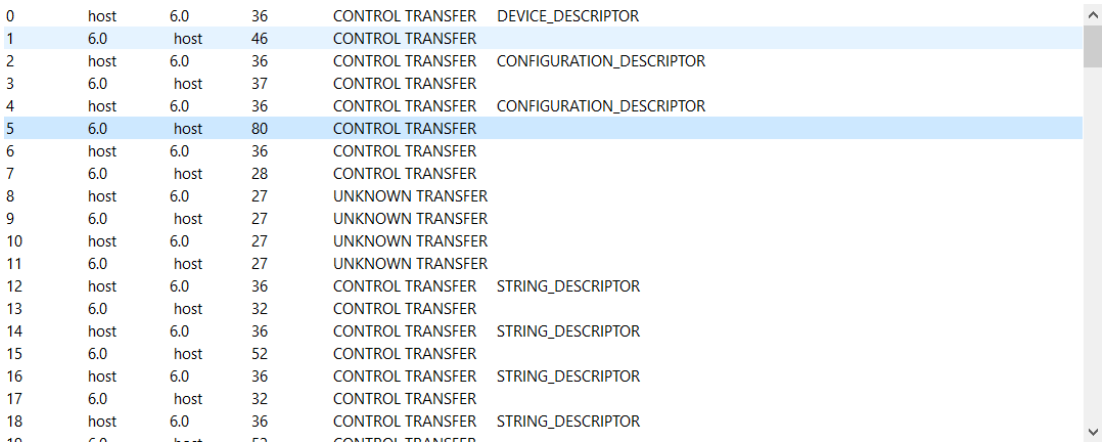
3.7 Zobrazenie základných informácií

V požiadavke **P10** sme si zadefinovali 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 najintutívnejšia 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. Takže by sme chceli vyobraziť jednotlivé pakety pomocou ich základných vlastností, ktoré by boli reprezentované v stĺpčekoch, pričom jeden riadok by reprezentoval konkrétny paket. Qt nám poskytuje 2 základné triedy pomocou ktorých budeme schopní jednoducho dosiahnuť žiadaného vzhľadu: *QListWidget* a *QTableWidget*.

QListWidget

QListWidget [67] poskytuje vzhľad na základe view triedy QListView [68] a jednotlivé položky sú reprezentované pomocou itemu QListWidgetItem [69]. 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



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

Obr. 3.7: Vyobrazenie základných informácií o pakete pomocou QListWidget

uchovávame práve v QByteArray, ktorý sme schopní zabaliť do QVariantu. Uloženie konkrétneho QVariantu je vykonané pomocou metódy `setData(int role, const QVariant &value)` – `role` môže byť podľa Qt dokumentácie [70] hodnota 256 a vyššie, druhý parameter je práve QVariant, 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 stĺpčeky. Tým, že jeden riadok je práve jeden item, stĺpčeky sú tu len akási abstrakcia, pretože všetky informácie ktoré môžeme vidieť na obrázku 3.7 (napríklad index paketu na začiatku alebo typ prenosu), je len jeden dlhý string. Z toho vyplýva, že ak by sme sa v budúcnosti rozhodli rozšíriť funkcionalitu nášho programu na triedenie itemov na základe hodnoty v jednotlivých stĺpčekoch, nebolo by to možné.

QTableWidget

QTableWidget [71] poskytuje vzhľad na základe view triedy QTableView [57] a jednotlivé položky sú reprezentované pomocou itemu QTableWidgetItem [72]. Pomocou QTableWidget sme dosiahli výsledok vyobrazený na obrázku 3.8 nižšie.

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 QTableWidgetItemami. To nám umožňuje pridať si samotný item pre každú informáciu o pakete, ktorú chceme vyobraziť, čím zároveň vytvárame skutočné stĺpčeky. QTableWidget má zároveň možnosť pridania horizontálnej hlavičky, v ktorej vieme špecifikovať význam hodnôt v jednotlivých stĺpčekoch.

S týmito vlastnosťami nám ale vznikli aj nové problémy, ktoré musíme riešiť. Prvý problém je, že nie všetky riadky majú jednotnú dĺžku (niektoré ne-

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

Obr. 3.8: Vyobrazenie základných informácií o pakete pomocou QTableWidget

majú 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 naštastie 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 uchovávame 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 nemáme jeden item reprezentujúci celý riadok, zafarbenie prebieha obyčajnou iteráciou cez všetky stĺpčeky a ich samostatným zafarbením. Celkovú funkcionalitu 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 QTableWidget

3.8 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 [73] trieda 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 vytvoriť 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 zobraziť 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 veiwportu alebo danej časti), mousePressEvent (kliknutie na viewport), atď.
- Používanie `viewport->update()` k updatenutiu kontentu na viewporte namiesto `update()`.

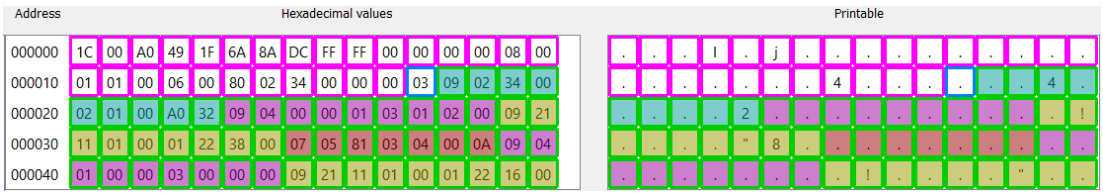
Poďme si ale rozmyslieť, či je skutočne nutné všetky tieto veci implementovať alebo či pre nás nebude výhodnejšie nájsť widget, ktorý to už má celú túto základnú funkcionalitu nadefinovanú a len ho upravíme našim požiadavkám hexdumpu.

QTableView

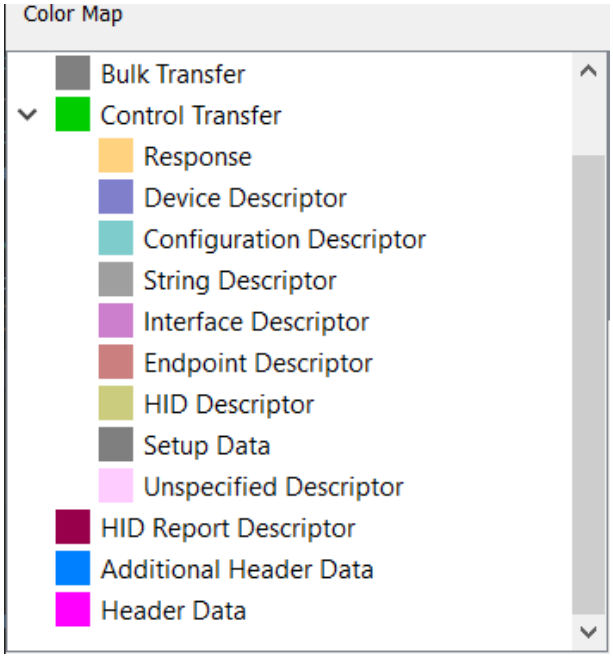
S QTableView [57] sme sa už stretli vyššie v sekcii 3.7. Hexdump si v podstate vieme predstaviť ako tabuľku dát, preto dáva zmysel skúsiť upraviť práve túto triedu. QTableView je súčasťou Qt model/view architektúry, takže nám umožňuje logicky oddeliť dáta a ich zobrazovanie. Na reprezentáciu dát máme vytvorený vlastný model, poďme 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 architektúre, na vykreslenie farebného oddelenia dát a zároveň na vyobrazenie dát samotných použijeme delegate (detailnejší popis jeho fungovania si takisto opíšeme v kapitole Vývojová dokumentácia 4). Ukážku nášho výsledného hexdumpu môžeme vidieť na obrázku 3.10 nižšie.



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



Obr. 3.11: Časť Color Map

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.

Address	Hexadecimal values																Printable															
000000	1C	00	A0	49	1F	6A	8A	DC	FF	FF	00	00	00	00	08	00	.	.	.	I	.	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

3.9 Zobrazenie sémantického významu dát

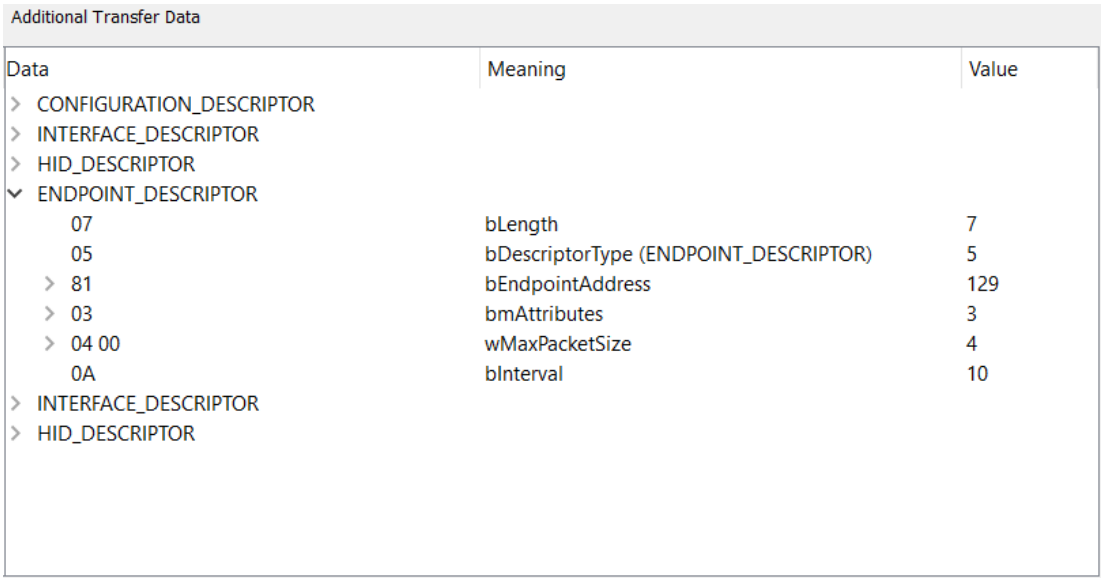
Ako sme si definovali v požiadavkách **P6**– **P8**, chceme byť schopní vykonávať sémenatikú analýzu pomocou stromovej štruktúry. Vzhľadom na to, že sa jedná o obecne známy spôsob vyobrazenia dát, Qt má na to špeciálnu triedu QTreeView [74], ktorá je sučasťou Qt model/view architektúry. Takisto máme opäť možnosť nadefinovať si vlastný view pomocou dedenia od triedy *QAbstractItemView* [75]. Ako sme už ale videli vyššie, museli by sme sami implementovať viaceré funkcionality, ktoré nám 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 architektúre, budeme sa teraz zaoberať iba vyobrazením dát. Keďže nám vyhovuje štandardný spôsob akým QTreeView zobrazuje svoje položky, definujeme len niekoľko základných vlastností pre jednotlivé viewery:

- Každému vieweru nastavíme jemu odpovedajúci model pomocou `setModel()`.
- Nastavíme šírku jednotlivých stĺpčekov buď pomocou `resizeColumnToContents()` alebo `setColumnWidth()`.

Príklad výslednej stromovej štruktúry môžeme vidieť na obrázku 3.13.



Obr. 3.13: Vyobrazenie stromovej štruktúry

4. Vývojová dokumentácia

Táto kapitola je zameraná pre programátorov, ktorých by bližšie zaujímali implementačné detaily a štruktúra nášho programu. Najprv si opíšeme ako skompilovať priložené zdrojové súbory práce a neskôr sa pozrieme na stavbu programu.

4.1 Kompilácia

V prílohe tejto práce sa nachádzajú zdrojové kódy nášho programu. Na ich úspešné skompilovanie budeme ale potrebovať splniť niekoľko požiadaviek:

- Podporovaná platforma na kompiláciu je momentálne iba Windows 10 verzie 20H2.
- Ku kompilácii je potrebné Visual Studio 2019 verzie 16.9.0, a Windows 10 SDK verzie 10.0.19041.0
- Qt verzie 5.15.0 [1]

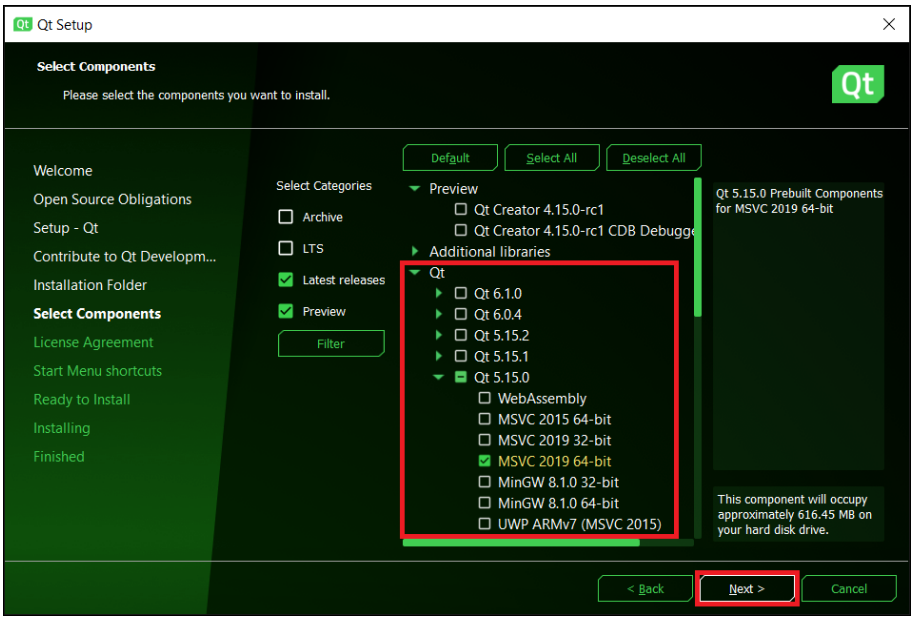
4.1.1 Inštalácia Qt

Keďže ku kompilácii budeme potrebovať samotné Qt, ukážeme si jeho inštaláciu v nasledujúcich krokoch (v prípade, že náš počítač už obsahuje požadovanú verziu Qt, môžeme tieto kroky preskočiť a prejsť na integráciu VS s Qt 4.1.2). Tu je nutné spomenúť, že sa jedná o online inštaláciu a budeme potrebovať pripojenie k internetu:

- Po kliknutí na odkaz https://www.qt.io/download sa nám otvorí internetový prehliadač so stránkou na stiahnutie Qt, kde vyhľadáme možnosť „Downloads for open source users“ a klikneme na „Go open source“.
- Na aktuálnej stránke nascrollujeme úplne dole, kde klikneme na zelené tlačidlo s nápisom „Download the Qt Online Installer“.
- Na aktuálnej stránke klikneme na „Download“, čím sa nám stiahne „Qt Online Installer“. Ten následne spustíme.
- Ako prvé nás privíta obrazovka, ktorá od nás vyžaduje prihlásenie sa do nášho Qt účtu. Toto je bohužiaľ nutnosť a nie je možné bez toho pokračovať. V prípade, že nemáme žiadny vytvorený Qt účet, môžeme si ho vytvoriť priamo počas inštalácie a celé to nezaberie viac ako dve minúty. Potom klikneme na tlačidlo „Next“.
- Teraz sa postupne preklikáme až po časť, kde máme možnosť si vybrať či chceme zasielať štatistické údaje počas používania Qt Creatoru. Tu si môžeme zvoliť ktorúkoľvek z možností podľa osobných preferencií a pokračovať ďalej v inštalácii.

[1] V dobe dokončovania práce bola najnovšia verzia Qt 6.1, ktorá by mala byť spätne kompatibilná, ale netestovali sme to.

- Postupne sa dostaneme na obrazovku kde si zvolíme konkrétne komponenty, ktoré sa nám nainštalujú. Komponenty, ktoré sú zaškrtnuté necháme tak a rozklikneme možnosť Qt←Qt 5.15.0 a zaškrtneme možnosť „MSVC 2019 64-bit“ a klikneme na „Next“ (obrázok 4.1).



Obr. 4.1: „Select Components“ časť Qt Setupu

- Ďalej nasledujú bežné kroky pomocou ktorých dokončíme inštaláciu.

4.1.2 Visual Studio 2019 a Qt

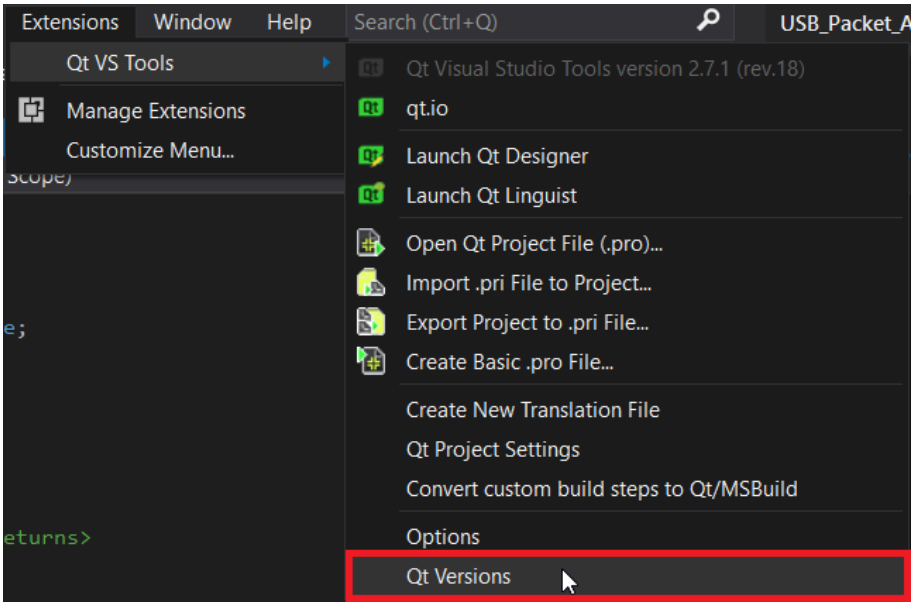
Teraz si ešte ukážeme integrovanie Visual studia 2019 spolu s Qt. Budeme si potrebovať nainštalovať *Qt VS Tools*. Podrobný postup si ukážeme v nasledujúcich krokoch:

- Priamo vo Visual Studiu si cez možnosť *Extensions→Manage Extensions* do online hľadania zadáme „Qt“ a stiahneme si rozšírenie „Qt Visual Studio Tools“ [76]

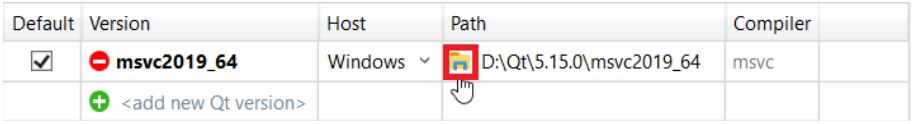
- Po úspešnej inštalácii si musíme ešte vybrať Qt verziu cez možnosť *Extensions→Qt VS Tools→Qt Versions* (ukázané na obrázku 4.2).

- V dialógu sa teraz cez ikonu v stĺpčeku „Path“ (obrázok 4.3) odkážeme do adresáru, kde sme nainštalovali *Qt* a následne do adresáru *5.15.0msvc2019_64bin* kde máme nainštalovaný *qmake.exe*.

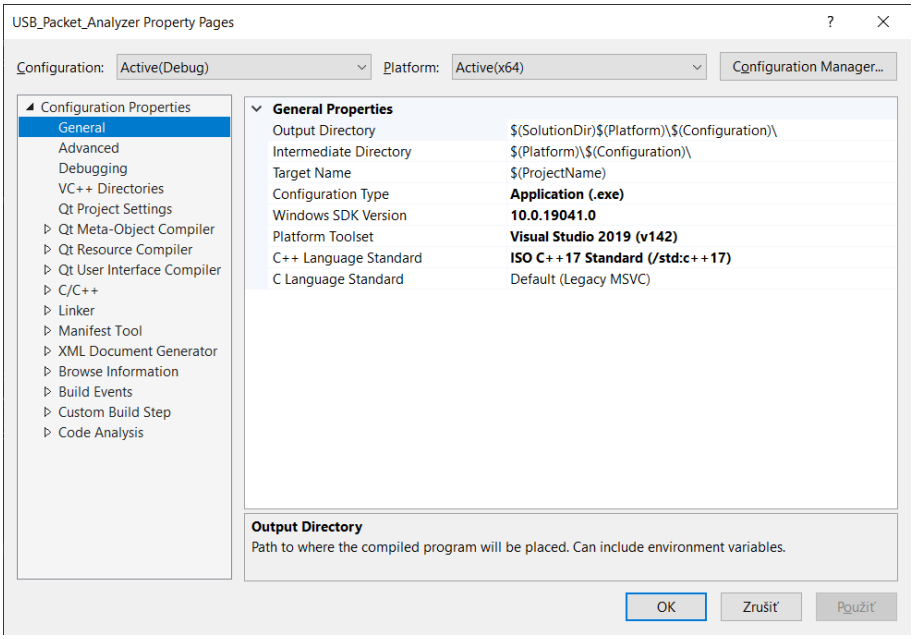
- Ako posledné si ešte skontrolujeme cez možnosť *Project→Properties* v *Configuration Properties→General* skontrolujeme, že máme nastavený *C++ Language Standard* na možnosť *ISO C++ 17* a *Windows SDK Version* na verziu *10.0.19041.0* ako na obrázku 4.4.



Obr. 4.2: Visual Studio možnosť vybratia Qt verzie.

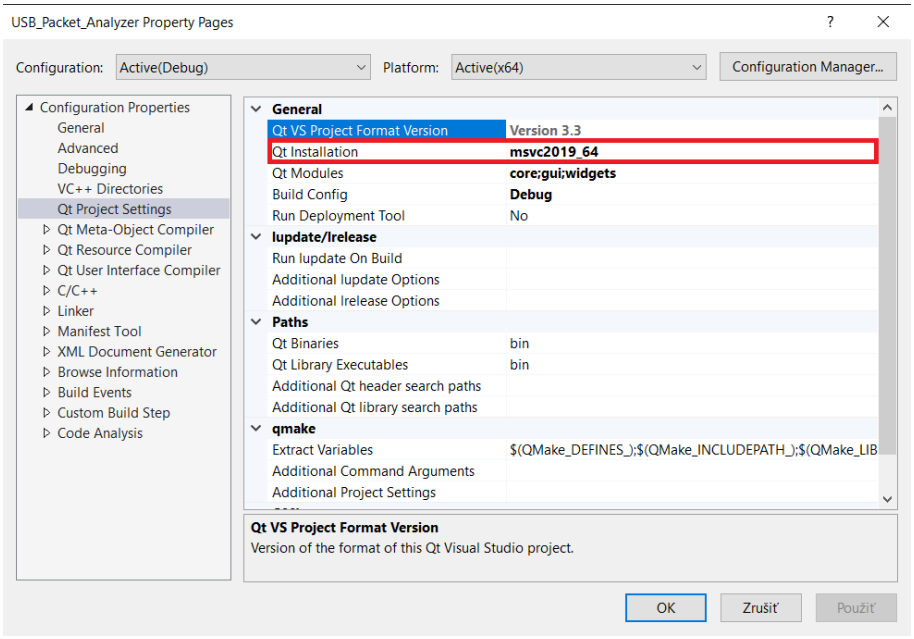


Obr. 4.3: Visual Studio zvolenie adresára ku *qmake.exe*.



Obr. 4.4: Visual Studio obecné nastavenia projektu.

Môže sa stať, že narazíme na bug keď nám krok 4.1.2 nenastaví Qt verziu a nebudeme vedieť projekt preložiť kvôli errorom typu „cannot open source file qbytearray“. V takom prípade prejdeme do možnosti *Project→Properties→ Qt Project Settings* a manuálne nastavíme *Qt Installation* na *msvc2019_64* (obrázok 4.5).



Obr. 4.5: Visual Studio manuálne nastavenie Qt Installation.

V tomto momente by sme mali byť schopní úspešne skompilovať a spustiť náš program.

4.1.3 Warningy pri kompilácii

Warningy prekladaču sú samozrejme veľmi dôležité a nemali by byť ignorované. Niekedy sa ale jedná o vedľajší efekt, ktorý upozorňuje na časti kódu o ktorých sme si vedomí a neželáme si ich meniť. Pri kompilácii prázdneho Qt programu nám prekladač vygeneruje niečo cez 250 warningov a preto sme sa rozhodli niektoré z nich vypnúť v nastaveniach projektu. V časti *Project→Properties→Configuration Properties→C/C++→Advanced* sme do položky *Disable Specific Warnings* pridali čísla nasledujúcich warningov, čím vypneme ich zobrazovanie:

C26812 [77] – „The enum type type-name is unscoped. Prefer 'enum class' over 'enum' (Enum.3)“ – tento typ warningu poukazuje aj na časti v našom kóde pretože používame tzv. „plain enum“ a porovnávame ho s položkami typu `int`, na ktoré majú plain enumy implicitnú konverziu.

Všetky nasledujúce warningy sa nachádzajú len v zdrojových kódoch Qt:

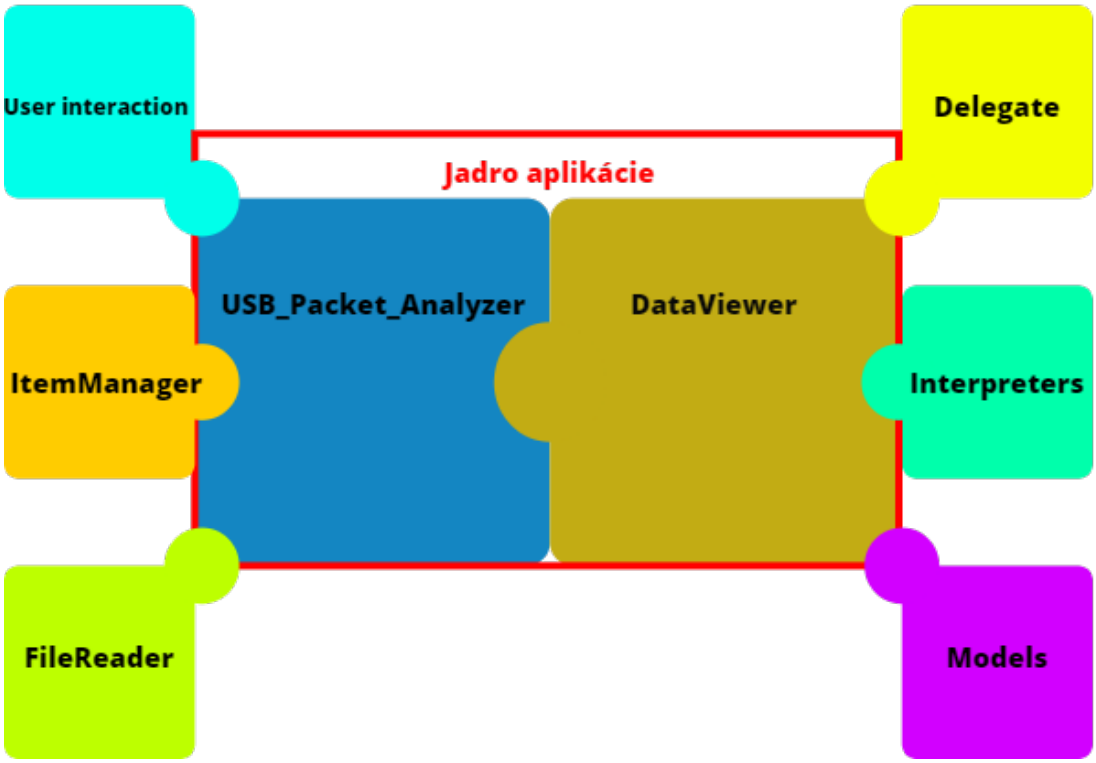
C26451 [78] – „Arithmetic overflow: Using operator 'operator' on a size-a byte value and then casting the result to a size-b byte value. Cast the value to the wider type before calling operator 'operator' to avoid overflow.“

C26498 [79] – „Use constexpr for values that can be computed at compile time.“

C26495 [80] – „Variable '%variable%' is uninitialized. Always initialize a member variable (type.6).“

C6011 [81] – „Dereferencing NULL pointer < name>.“

4.2 Architektúra aplikácie



Obr. 4.6: Diagram architektúry aplikácie

V tejto sekcii si prejdeme celkovú stavbu aplikácie a akým spôsobom sú jednotlivé komponenty prepojené. Ako vidíme na obrázku 4.6, aplikáciu môžeme rozdeliť do dvoch hlavných komponent, ktoré tvoria logické jadro programu: `USB_Packet_Analyzer` a `DataViewer`, na ktoré sa viažu ďalšie komponenty a dopĺňajú ich funkcionalitu. Teraz si podrobnejšie opíšeme obe hlavné komponenty spolu s komponentami, ktoré sú na nich napojené.

4.3 USB_Packet_Analyzer

USB_Packet_Analyzer tvorí hlavnú triedu programu, ktorá reprezentuje okno zobrazené užívateľovi hneď po zapnutí aplikácie. Naše hlavné okno pozostáva z nasledujúcich komponent:

- QTableWidget – slúži na vyobrazenie základných informácií o paketoch.
- 2 QRadioButtony – slúžia na výber medzi analýzou fixného súboru (file capture), alebo súboru do ktorého môže sniffer počas analýzy niečo pripísať (live capture).

- QCheckBox – slúži na výber, či užívateľ chce farebne zvýrazniť detailnejší význam analyzovaných paketov alebo nie.
- QLabel – slúži na vyobrazenie názvu súboru, ktorý sa aktuálne spracováva.
- QProgressBar – progress bar ukazujúci stav spracovania súboru počas file capture.
- 5 QPushButtonov – tlačidlá, ktorých jednotlivú funkcionalitu si vysvetlíme nižšie v tejto sekcii 4.3.1.

Zároveň implementuje funkcie spojené s užívateľskou interakciou, od ktorej sa následne odvíja ďalšie správanie aplikácie.

4.3.1 Užívateľská interakcia

Ako sme už spomínali vyššie v kapitole 3.3, Qt využíva na komunikáciu medzi objektami tzv. „Signals & Slots“ [58] mechanizmus, ktorý funguje ako alternatíva ku callback funkciám v iných frameworkoch. Qt widgety majú veľa preddefinovaných signálov (ku ktorým si môžeme dodefinovať ďalšie), ktoré sú emitované pri konkrétnom evente (napríklad QPushButton [82] má signál `clicked()` ktorý je emitnutý pokiaľ je dané tlačidlo stlačené). Slot je funkcia, ktorá je zavolaná ako odpoveď na konkrétny signál. Prepojenie signálu so slotom prebieha pomocou metódy `QObject::connect(QObject* sender, signal, QObject* receiver, method)`, kde postupne definujeme inštanciu QObject spolu s jej konkrétnym signálom a následne inštanciu QObject spolu s jej metódou, ktorá sa má zavolať po emitovaní daného singálu. Qt má takisto možnosť „Auto-Connect“ [83] pri ktorej stačí, že sa budeme držať štandardných konvencií a tým pádom nebudeme musieť manuálne prepájať jednotlivé signály a sloty pomocou `connect()` metódy. Toto využívame napríklad pri tlačidlách, konkrétne so signálom `clicked()`, kde stačí ak si vytvoríme slot pomocou nasledujúcej mennej konvencie: `void on_<object name >_clicked()`.

Obr. 4.7: Tlačidlá v aplikácii.

Ako môžeme vidieť na obrázku 4.7, naša aplikácia obsahuje hneď niekoľko tlačidieľ s odlišnou funkcionalitou o ktorú sa starajú už jednotlivé sloty, ktoré si teraz postupne opíšeme.

Open tlačidlo slúži na vybratie si súboru na analýzu. To funguje na základe triedy QFileDialog [84], ktorá slúži na prechádzanie file systému. Užívateľovi sa zobrazí nový dialóg s intuitívnym ovládaním na ktoré je bežný užívateľ zvyknutý. Zároveň slúži na resetovanie progress baru a nastavenie labelu, ktorý zobrazuje aktuálne zvolený súbor.

Start tlačidlo ako už napovedá jeho samotný názov, slúži na začatie analýzy. V tejto chvíli sa začne spracovávať súbor, ktorý si užívateľ zvolil pomocou predchádzajúceho tlačidla Open 4.3.1. Spracovanie je následne vyobrazené pomocou QTableWidgetu, ktorým ako sme už riešili v kapitole 3.7 vyobrazujeme základné informácie o jednotlivých paketoch. O cekové spracovanie súboru a jeho vyobrazenie tak ako aj o iné veci, sa stará trieda *ItemManager*, o ktorej si povieme viac neskôr v sekcii 4.3.2.

Clear tlačidlo zastáva funkciu vyčistenia plochy QTableWidgetu na ktorej sú vyobrazné základné informácie o paketoch. Takisto resetne progress bar, ktorým reprezentujeme v akom stave sa nachádzame z hľadiska spracovania daného súboru.

Stop tlačidlo spôsobí to, že sa úplne zastaví spracovávanie súboru. Využívané je pri analýze paketov v reálnom čase, čím stopne pridávanie nových paketov do QTableWidgetu. Akékoľvek ďalšie spracovávanie súborov už nie je umožnené. Ak by sme chceli analýzu len pozastaviť, tak v takom prípade použijeme tlačidlo Pause 4.3.1.

Pause tlačidlo ako už bolo naznačené vyššie, slúži na pozastavenie analýzy. To znamená, že nové pakety nie sú pridávané do QTableWidgetu. Obnovenie analýzy je možné opätovným stlačením tlačidla (teraz už s nápisom „Continue“), čím bude obnovené pridávanie nových paketov.

Z užívateľskej interakcie máme k dispozícii ešte jednu, o ktorej sme viac hovorili v kapitole 3.7, kde sme sa pre splnenie požiadavky na vyobrazenie detailnejších informácií o pakete na základe užívateľskej interakcie rozhodli pre dvojklik na položku reprezentujúcu daný paket.

`on_tableWidget_itemDoubleclicked(QTableWidget* item)` je slot, ktorý sa volá po dvojkliku na konkrétny item v QTableWidget (pointer na daný item je poslaný ako parameter). V tejto metóde vytvoríme novú inštanciu triedy DataViewer s odpovedajúcimi parametrami a vyobrazíme jej dialógové okno, ktoré zobrazí detailnejšie informácie o danom pakete.

4.3.2 ItemManager

Interakciu užívateľa so systémom sme si už vysvetlili a teraz sa poďme pozrieť na samotné spracovávanie súborov. To má na starosti hlavne trieda ItemManager. Tá je implementovaná podľa návrhového vzoru singleton a vytvárame ju v konštruktore hlavnej triedy USB_Packet_Analyzer. Tento návrh nám okrem iného zároveň umožňuje počas analýzy v reálnom čase pokračovať v spracovávaní súboru na mieste, kde sme predtým skončili. V kapitole 3.4 sme sa rozhodli, že na čítanie súborov budeme používať QFile. ItemManager priamo so súborom, ale už len predparsovanými dátami v podobe jednotlivých paketov. Tie mu posiela inštancia triedy FileReader, ktorú má uloženú ako dátovú položku. FileReader má implementovanú celú logiku práce so súborom, pričom asi najpodstatnejšia je metóda `GetPacket()` – prečíta zo súboru dáta, ktoré reprezentujú jeden paket a tie vráti zabaené v QByteArray.

Selekcia paketov

Ako sme už spomínali vyššie, samotná analýza začína stlačením tlačidla Start, kedy sa na inštancii ItemManagera zavolá metóda `ProcessFile(QString filename, bool liveReading)`. Ako nám už samotné názvy parametrov napovedajú, predávame v nej názov súboru, ktorý chceme spracovať a `bool` hodnotu určujúcu, či sa jedná o live capture alebo file capture. Táto metóda zaistí prostredníctvom FileReader inštancie otvorenie daného súboru a prečítanie jeho globálnej hlavičky. Ak sa jedná o live capture, nastaví timer aby kontroloval zmenu v súbore v pravidelných intervaloch 1 sekundy. Následne zavolá metódu `ProcessFileTillEnd(bool liveReading)`, ktorá pokračuje v spracovaní súboru od miesta kde čítanie naposledy skončilo, až po jeho koniec. Táto metóda je takisto spojená s `QTimer::timeout()` signálom, pre dodatočné spracovanie súboru v prípade pripísania nových paketov počas live capture. V tejto metóde si postupne od FileReaderu pýtame pomocou `GetPacket()` metódy dáta reprezentujúce jednotlivé pakety a tie posielame ako parameter funkcie `ProcessPacket(QByteArray)` na ich ďalšie spracovanie. Spracovanie samostatného paketu obsahuje niekoľko fáz:

Kontrola pre špeciálny typ paketu.
Pri spracovaní jednotlivých paketov riešime 2 nasledujúce veci:

- Dáta paketu reprezentujú Configuration Descriptor spolu s Interface/HID/Endpoint Descriptormi – v takomto prípade bolo na zbernicu pripojené nové zariadenie a je potrebné si pre neho vytvoriť novú inštanciu typu BusDevice, o ktorej si viac povieme neskôr 4.7.1.
- Dáta paketu reprezentujú HID Report Descriptor – v takom prípade je nutné daný Report Descriptor rozparsovať. Detailnejší opis si povieme nižšie v sekcii 4.5.

Po týchto dvoch kontrolách a prípadne následných úkonoch, ktoré s nimi súvisia, posunieme dáta do metódy `FillUpItem(QByteArray)`.

Rozdelenie dát do logických celkov.
Jej hlavná úloha je rozdeliť celý QByteArray na niekoľko menších podľa ich logického významu – hlavička paketu, dodatočné dáta hlavičky a zvyšok poslaných dát. V kapitole 3.5 sme sa rozhodli, že všetky tieto dáta zabalíme do QVariantu a uložíme do prvého QTableWidgetu na aktuálnom riadku. Ale ešte predtým ako dáta uložíme, si musíme celý riadok s odpovedajúcimi QTableWidgetami vytvoriť.

Pridanie jednotlivých riadkov.
Na to nám slúži metóda `InsertRow(PUSBPCAP_BUFFER_PACKET_HEADER, unsigned char*)` – prvý parameter je pointer na štruktúru reprezentujúcu hlavičku paketu, druhý parameter reprezentuje char* na dáta paketu. Pomocou tejto metódy teda vytvárame jednotlivé riadky v našej QTableWidget komponente. Po pridaní jednotlivých QTableWidgetItemov sa zavolaním metódy `ColorRow(PUSBPCAP_BUFFER_PACKET_HEADER)` postará o farebné oddelenie jednotlivých riadkov podľa ich významu. Po úspešom pridaní riadku do QTableWidgetu a naplnení uložení jednotlivých dát do prvého QTableWidgetItemu sa

vraciame späť do metódy `ProcessPacket(QByteArray)`, kde musíme urobiť ešte poslednú vec – skontrolovať, či daný paket nereprezentuje Setup Paket pomocou metódy `CheckForSetupPacket(QByteArray)`.

Kontrola na Setup Paket.
Ak sa jedná o Setup Paket, skontrolujeme položku bRequest. Ak má hodnotu GET_DESCRIPTOR, znamená to, že USB host si od zariadenia vypýtal konkrétny typ desrcriptoru – ten si vieme zistiť z položky wValue. V prípade, že sa jedná o Configuration Descriptor alebo HID Report Descriptor tak vieme, že dáta nasledujúceho paketu budú reprezentovať tento konkrétny descriptor a teda bude nutné ho podľa toho spracovať.

ItemManager obsahuje ešte jednu dôležitú metódu, a to `GetDataType(QTableWidgetItem*, QTableWidgetItem*)`, ktorá je schopná na základe dát, ktoré sme si uložili do konkrétneho QTableWidgetItemu zistiť presný typ prenosu. V prípade, že sa jedná o Control Transfer tak určuje aký typ descriptoru reprezentujú „zvyškové dáta“ paketu (dáta nasledujúce po hlavičke a po nepovinnej dodatočnej hlavičke). V danom prenose sa ale môže nachádzať aj viac descriptorov súčasne – konkrétne je to v prípade ak zariadenie posiela svoj Configuration Descriptor spolu s Interface/Endpoint/HID descriptormi. V tomto prípade metóda vráti, že sa jedná o Configuration Descriptor.

Týmto sme si opísali akým spôsobom sa spracovávajú súbory, vyobrazujú sa základné informácie o paketoch a spôsob uloženia dát jednotlivých paketov. Teraz sa pozrieme ako je v našom programe riešená detailnejšia analýza (napírklad hexdump alebo sémantické vyobrazenie dát) konkrétnych paketov na základe informácií, ktoré sme si pri nich uložili.

4.4 DataViewer

Trieda reprezentujúca pop-up okno s detailnejšou analýzou konkrétneho paketu. Dané okno pozostáva z nasledujúcich komponent:

- 2 QTableView – slúžia na vyobrazenie hexdumpu. Jeden pre hex časť a druhý pre tlačiteľné znaky.
- 3 QTreeView – každý slúži na vyobrazenie sémantického významu rozdielnych dát v podobe stromovej štruktúry. Jeden zobrazuje hlavičku paketu spolu s nepovinnou hlavičkou, ďalší vyobrazuje Color Map slúžiacu na lepšiu orentáciu v hexdumpe a posledný ukazuje zvyškové dáta – napríklad význam rôznych descriptorov alebo inputu zariadenia.
- 6 QLabelov – slúžia na opis toho, čo je vyobrazené na predošlých komponentách.

Ako sme sa si vysvetlili vyššie 4.3.1, nová inštancia DataVieweru sa vytvára v momente keď užívateľ dvojklikne na riadok reprezentujúci konkrétny paket. Do konštruktoru si predávame 3 základné veci:

1. Pointer na QTableWidgetItem, ktorý v sebe obsahuje dáta daného paketu.

2. Typ prenosu/descriptoru, ktorý získame pomocou vyššie sponínanej metódy `ItemManager::GetDataType(QTableWidgetItem*, QTableWidgetItem*)`.

3. Bool hodnotu, ktorá určuje či budeme dáta v hexdumpe farebne zvýrazňovať alebo nie. Túto hodnotu získame z QCheckBoxu hlavného okna.

V konštruktore následne inicializujeme naše komponenty – keďže využívame spôsob programovania pomocou Qt Model/View architektúry, inicializácia zahŕňa vytvorenie modelov a delegátov a ich následné priradenie k jednotlivým viewerom. Teraz si povieme obecný spôsob fungovania modelov a delegátov v Qt, a následne si priblížime implementáciu našich konkrétnych modelov.

4.4.1 Model

Model je v Qt obyčajná trieda založená na QAbstractItemModel [**?**] triede, ktorá spĺňa preddefinovaný interface vďaka ktorému viewery a delegáti pristupujú k dátam. Naše modely môžu byť takisto odvodené od viac špecifických modelov, ktoré poskytuje Qt ako napríklad QAbstractTableModel [85]. Nezáleží na tom akým spôsobom budú naše dáta uložené, modely ich reprezentujú ako hierarchickú štruktúru obsahujúcu tabuľku itemov. Prístup k položkám modelu je potom zaistený pomocou tzv. „model indexov“. Viewery a delegáti používajú tieto indexy aby sa odkázali na konkrétne dáta, ktoré neskôr vyobrazujú. Aby sme získali index na konkrétne dáta, potrebujeme špecifikovať tri veci: číslo riadku, číslo stĺpca a model index rodičovského itemu.

Číslo riadku a číslo stĺpca: Ako sme už spomínali, modely reprezentujú itemy pomocou tabuľkovej štruktúry. Pomocou čísla riadku a stĺpca sa odkážeme na konkrétny item v tabuľke.

Model index rodičovského itemu: Tabuľková štruktúra ponúka dobrú reprezentáciu v prípade tabuliek, avšak nie je úplne ideálna ak si dáta ukladáme a chceme ich vyobraziť v podobe stromovej štruktúry. V takejto štruktúre má prirodzene každý item (až na koreň stromu) rodiča a ľubovoľný počet synov. Na index konkrétneho vrcholu v strome nám tak bude stačiť poznať index jeho rodiča a ktorý syn v poradí to je (to môžeme reprezentovať číslom riadku). Každý item v modeli má definovanú tzv. „role“, ktorá určuje ako sa na dané dáta máme pozerať. Napríklad *Qt::DisplayRole* označuje dáta, ktoré majú byť vo vieweri vyobrazené ako text

4.4.2 Delegát

Delegáti obecne slúžia na vizualizáciu dát s možnosťou ich editácie. Štandardný interface pre delegátov je poskytnutý triedov QAbstractItemDelegate [86]. Od delegátov sa predpokladá, že implementujú metódy `paint()` a `sizeHint()` na vyobrazenie kontentu.

4.4.3 Hexdump

Teraz sa pozrieme na konkrétny model pre naše komponenty reprezentujúce hexdump. Implementujeme ho vlastnou triedou `HexdumpModel`, ktorá je odvodená od `QAbstractTableModel`. V konštruktore si predávame nasledujúce:

1. `QTableWidgetItem* item` – Pointer na item obsahujúci dáta konkrétneho paketu, ktoré budeme v hexdumpe vyobrazovať.

2. `bool hexView` – Keďže máme rovnaký model pre hex časť a tlačiteľné znaky, potrebujeme vedieť rozlíšiť o ktorú z týchto dvoch možností sa jedná. Na to presne slúži táto bool hodnota.

3. `HeaderDataType additionalDataType` – Typ prenosu/descriptoru dát paketu.

4. `QObject* parent` – Pointer na DataViewer inštanciu.

QAbstractTableModel ponúka viac špecializovaný interface ako QAbstractItemModel a podľa dokumentácie implementuje metódy `index()` a `parent()`. Nám ostáva implementovať metódy `columnCount()`, `rowCount()` a `data()`. Odporúčané je takisto implementovať metódu `headerData()`.

columnCount() metóda vráti počet stĺpcov z ktorých bude pozostávať výsledný hexdump. V našom programe sme sa rozhodli zvoliť konštantnú hodnotu, ktorá je pre hexdump bežná – 16.

rowCount() metóda vráti počet riadkov, z ktorých bude pozostávať výsledný hexdump. Vzhľadom na to, že máme k dispozícii QTableWidgetItem obsahujúci všetky dáta, ktoré budeme chcieť vyobraziť v hexdumpe, stačí nám spočítať ich dĺžku a vydeliť ju počtom stĺpcov na jeden riadok (pričom z výsledku berieme hornú celú časť).

data() metóda vracia QVariant reprezentujúci konkrétny item v hexdumpe. Ten pozostáva z dvojice:

1. Konkrétne dáta, ktoré budú vyobrazené – 1B buď v hex tvare, alebo v tvare tlačiteľného znaku

2. `HeaderDataType` položka určujúca presný typ prenosu/descriptoru. Podľa tejto položky budeme následne určovať akou farbou bude daný item v hexdumpe označený. Nemôžeme tu ale iba skopírovať položku, ktorú sme dostali ako parameter v konštruktore, pretože hodnota tejto položky pochádza z metódy `ItemManager::GetDataType` a ako sme už spomínali, tá nevie rozoznať rozdiel medzi jednotlivými descriptormi pokiaľ sú poslané v jednom prenose. My ale chceme vedieť farebne odlíšiť aj descriptory, ktoré zariadenie poslalo v jednom prenose. Konkrétnu hodnotu teda zistíme pomocou metódy `GetDataRepresentationType` v ktorej pokiaľ sa jedná o Configuration Descriptor, prejdeme zvyškové dáta paketu a na základe indexu v ktorej časti sa nachádzame presne určíme typ descriptoru.

headerData() metóda nám umožňuje nastaviť vertikálnu hlavičku na ktorú je užívateľ pri bežnom hexdumpe zvyknutý – offset bytov v hexadecimálnej reprezentácii.

4.4.4 Hexdump delegát

Delegát pre hexdump je naša trieda `HexdumpDelegate` odvodená od triedy QStyledItemDelegate. Tento delegát slúži na vyobrazenie jednotlivých položiek v hexdumpe, tak ako aj ich farebné zvýraznenie. Pri vytváraní inštancie daného delegátu jej nastavíme bool parameter *dataHighlight* určujúci, či máme farebne zvýrazňovať dáta v hexdumpe. Ako sme už naznačili vyššie v sekcii 4.4.2, musíme implementovať metódy `paint()` a `sizeHint()`.

paint() metóda dostane ako parameter model index z ktorého získa dvojicu dát QVariant, ktorú sme vytvorili pomocou nášho hexdump modelu v metóde `data()`. Podľa dátovej položky *dataHighlight* sa určí, či budeme farbne zvýrazňovať daný item – ak áno, z druhej položky QVariantu získaného z model indexu zistíme presný typ prenosu/descriptoru (teraz už aj v prípade, že bolo poslaných viac descriptorov v jednom prenose) a na základe toho zafarbíme daný item.

sizeHint() metódou vrátime konštantnú hodnotu `QSize(30,30)` ktorá určuje veľkosť bunky hexdumpu.

Teraz keď sme si vysvetlili fungovanie HexdumpModelu a HexdumpDelegatu, stačí ak ich nastavíme danému QTableVieweru. To urobíme pomocou metódy `setModel()` a `setItemDelegate()`.

Momentálne nám ostávajú ešte modely pre 3 komponenty reprezentujúce Color Map, hlavička paketu a zvyškové dáta. Všetky majú viewer typu QTreeView, takže k nim budeme potrebovať model prispôsobený na stromovú štruktúru. Každá komponenta má vlastný model, niektorú funkcionalitu ale majú spoločnú a preto sme si vytvorili triedu `TreeItemBaseModel`, ktorá dedí od triedy `QAbstractItemModel`. Každý z modelov pre jednotlivé QTreeViewery bude následne dediť od `TreeItemBaseModel`. Musíme ešte vyriešiť spôsob reprezentácie tejto stromovej štruktúry. Na to nám poslúži naša trieda `TreeItem`. Pri implementácii tried TreeItemBaseModel a TreeItem sme sa inšpirovali príkladom z Qt dokumentácie [87].

4.4.5 TreeItem

Reprezentuje jeden vrchol v stromovej štruktúre. Uchováva si v sebe nasledovné:

- `TreeItem* parent` – pointer na svojho rodiča.
- `QVector< std::shared_ptr< TreeItem»childs` – list svojich synov
- `QVector< QVariant>` data – dáta, ktoré budú následne vyobrazené v QTreeView. Viewer samotný ponúka možnosť viacerých stĺpcov v jednom iteme a preto si dáta ukladáme v QVectore. Každá položka QVariant vo vectore tak rerezentuje dáta v jednotlivých stĺpcoch.

Následne tu máme implemtované bežné funkcie potrebné pre prácu so stromovou štruktúrov ako napríklad `ChildCount()`, `AppendChild()` alebo `ColumnCount()`, ktorých význam a princíp fungovania je bližšie opísaný v zdrojvých kódoch v súboroch *TreeItem.hpp* a TreeItem.cpp, ktoré boli poskytnuté ako príloha k tejto práci. Trochu podstatnejší je konštruktor, ktorý vyzerá nasledovne: `TreeItem(QVector< QVariant>data_, TreeItem* parent_)` – ako parameter dostávame dáta, ktoré si bude daný vrchol držať a pointer na svojho rodiča. V prípade že sa jedná o koreň stromu bude rodič nastavený na `nullptr`.

4.4.6 TreeItemBaseModel

Poskytuje spoločný interface pre modely fungujúcimi pre stromové štruktúry. Takisto v sebe drží `std::unique_ptr< TreeItem>rootItem` reprezentujúci koreň stromu. Ako spoločný predok všetkých modelov pre stromové štruktúry má implementované metódy, ktorých funkcionalita sa v odvodených triedach nemení a patria medzi ne:

- `index()` – vytvorí a vráti QModelIndex na základe parametrov (číslo riadku, číslo stĺpca a model index rodičovského itemu).
- `parent()` – na základe parametru QModelIndex vráti index rodiča daného vrcholu. Ak sa jedná o koreň stromu, podľa konvencie vrátime QModelIndex().
- `rowCount()` – vráti počet synov vrcholu.
- `columnCount()` – vráti počet stĺpcov dát vrcholu.

Následne tu máme ešte implementované metódy, ktoré sa nám hodia pri vyobrazovaní dát v QTreeView ale nie sú súčasťou QAbstractItemModelu:

- `CharToHexConvert(char** addr, int len, QString& data)` – v parametri *data* vráti hex reprezentáciu prvých *len* znakov char poľa uloženého na adrese *addr*.
- `ShowBits< T>(uint32_t start, size_t size, T number)` – parametrizovaná funckia, ktorá bola vytvorená pre účel splnenia požiadavky **P9** na vyobrazenie významu dát až na úrovni jednotlivých bitov. Výsledok zapisujeme do QStringu, ktorý na konci vrátime. Ako parameter T dostaneme číslo, ktorého bity budeme chcieť ukázať. To prechádzame postupne bit po bite a v momente keď sa nachádzame v intervale < start, start + size) zapisujeme do QStringu hodnoty jednotlivých bitov. Ak sa nachádzame mimo intervalu, zapisujeme „.“.

TreeItemBaseModel obsahuje ešte tzv. „pure virtual“ funkcie `data()` a `headerData()`, ktorých implementácia sa bude líšiť na základe dedenej triedy – je teda ich povinnosťou tieto metódy implementovať.

4.4.7 ColorMapModel

ColorMapModel je naša trieda odvodená od TreeItemBaseModel reprezentujúca model pre Color Map. Ako sme už vyššie spomínali, musí implementovať 2 funkcie: `headerData()` a `data()`. Ešte predtým si ale musíme rozmyslieť odkiaľ zoberieme dáta, ktoré budeme chcieť v Color Mape zobrazovať. Vyššie sme si spomínali triedu `TreeItem` 4.4.5, ktorá reprezentuje jeden vrchol v stromovej štruktúre, ktorá si bude udržiavať naše dáta. Celý strom si musíme najprv vytvoriť a naplniť ho dátami, a o to sa stará funkcia `SetupModelData()`.

SetupModelData() je metóda, ktorú voláme priamo v konštruktore ColorMapModel. Keďže dedíme od triedy TreeItemBaseModel, máme k dispozícii koreň stromu (`std::unique_ptr< TreeItem>rootItem`) pomocou ktorého budeme cez TreeItem metódy budovať strom. Dáta každého vrcholu budú pozostávať z 2 položiek:

- `HeaderDataType` – položka určujúca presný typ prenosu/descriptoru. Vďaka tejto položke budeme schopní priradiť odpovedajúcu farbu každému itemu v QTreeView.
- string reprezentujúci názov typu prenosu/descriptoru.

ColorMapModel obsahuje ešte jednu špecifickú metódu `GetDataTypeColor(HeaderDataType dataType)`, ktorá na základe parametru type vráti jemu odpovedajúcu farbu. Farby ku každému HeaderDataType máme nadefinované v triede DataHolder o ktorej si povieme viac nižšie 4.6. Teraz si prejdeme implementáciu už spomínaných virtuálnych funkcií:

- `headerData()` – keďže v Color Map nechceme mať žiadnu hlavičku, funkcia vracia prázdny QVariant().
- `data(QModelIndex& index, int role)` – podľa parametru role zistíme o aké dáta sa jedná. Ak sa role rovná *Qt::DecorationRole*, vrátime farbu pomocou metódy GetDataTypeColor(). Ak sa role rovná *Qt::DisplayRole*, vrátime textovú reprezentáciu typu prenosu/descriptoru, ktorú sme si uložili do itemu ako druhú položku. Ak sa role rovná hocičomu inému, vrátime prázdny QVariant().

4.4.8 USBPCapHeaderModel

je trieda odvodená od TreeItemBaseModel a reprezentuje model pre QTreeView, ktorý vyobrazuje detailnejšie informácie hlavičky paketu. V tejto triede máme takisto metódu `SetupModelData()`, ktorá vytvorí stromovú štruktúru reprezentujúcu hlavičku paketu. Hlavička paketu má fixný formát určený štruktúrou `USBPCAP_BUFFER_PACKET_HEADER`, ktorú definuje sniffer USBPcap cez ktorý zachytávame pakety na zbernici. Virtuálne funkcie sú implementované nasledovne:

- `headerData(int section, Qt::Orientation orientation, int role)` – dáta hlavičky si typicky budeme ukladať do QVectoru *data* nášho rootItemu. Podľa parametru orientation zistíme o akú hlavičku sa jedná

(vertikálnu/horizontálnu). Keďže v našom QTreeView chceme mať horizontálnu hlavičku, testujeme na zhodu práve s Qt::Horizontal hodnotou a zároveň kontrolujeme či sa role rovná Qt::DisplayRole. Následne už len vrátime dáta zo stĺpca, ktorý je definovaný parametrom section.

- `data(QModelIndex& index, int role)` – pokiaľ sa role nerovná Qt::DisplayRole alebo index nie je validný, vrátime prázdny QVariant(). V opačnom prípade is pomocou indexu získame pointer na daný item a vrátime dáta z konkrétneho stĺpca (hodnotu stĺpca zistíme takisto z indexu).

4.4.9 AdditionalDataModel

je trieda, ktorá je takisto odvodená od TreeItemBaseModel a reprezentuje model zvyškových dát paketu. To zahŕňa napríklad dáte reprezentujúce rôzne typy descriptorov, input zariadení atď. Virtuálne metódy `data()` a `headerData` riešime rovnakým spôsobom ako pri USBPCapHeaderModel 4.4.8. Problém nastáva až vo funkcii, ktorá má vytvoriť stromovú štruktúru. V minulých modeloch sme totiž presne vedeli čo majú jednotlivé dáta reprezentovať a ako bude vyzerať im odpovedajúci strom. Teraz to ale nie je také jednoduché, pretože zvyškové dáta paketu môžu mať rozličnú reprezentáciu, ktorá je určená až za runtimu. (odkaz na cieľ s ľahkou rozsiritelnostou ?) Preto túto situáciu vyriešime pomocou známeho návrhového vzoru – factory. Zadefinujeme si triedy, ktorých jediná úloha bude vytvoriť špecifickú stromovú štruktúru odpovedajúcu danému typu dát. Tieto triedy budeme obecne nazývať „Interpretery“.

4.4.10 BaseInterpreter

Je základná trieda od ktorej sú všetky Interpretery odvodené. Obsahuje nasledujúce dátové položky spoločné pre všetky Interpretery:

- `TreeItem* rootItem` – koreň stromu.
- `QTableWidgetItem* item` – pointer na item, ktorý v sebe obsahuje dáta z ktorých budeme zostavovať daný strom.
- `AdditionalDataModel* additionalDataModel` – pointer na model pre zvyškové dáta. Budeme ho potrebovať k využívaniu funkcií `ShowBits()` a `CharToHexConvert()`, ktoré sa nám hodia pri vyobrazovaní informácií v QTreeView.

Trieda má zadefinovaný konštruktor, ktorý má ako parameter všetky vyššie spomínané dátové položky. Zároveň obsahuje pure virtual funckiu `Interpret()`, ktorú si definuje každý interpreter zvlášť a slúži na vytvorenie konkrétnej stromovej štruktúry odpovedajúcej danému interpreteru.

Teraz si prejdeme všetky definované Interpretery a akú majú funckionalitu:

ConfigDescriptorsInterpreter
Slúži na interpretovanie nasledujúcich descriptorov:

- Configuration Descriptor

- Other_Speed_Configuration Descriptor
- Interface Descriptor
- HID Descriptor
- Endpoint Descriptor
- unknown descriptor – descriptor, ktorého štruktúru nepoznáme. Obecne ale vieme, že prvé 2B každého descriptoru reprezentujú položky *bLength* a *bDescriptorType* takže ich vypíšeme a ako ďalší vrchol stromu pripojíme zvyšné dáta descriptoru.

Keďže Interface/HID/Endpoint/unknown descriptory sú poslané v Control Transfere po Configuration Descriptore/Other_Speed_Configuration Descriptore, tak prechádzame celé zvyškové dáta paketu a postupne staviame strom pre jednotlivé typy descriptorov.

DeviceDescriptorInterpreter a DeviceQualifierDescriptorInterpreter
Samostatné triedy, ktoré slúžia na interpretovanie Device Descriptoru a Device_Qualifier Descriptoru. Oba descriptory majú zo začiatku rovnakú štruktúru a preto máme metódu `InterpretControlTransferDeviceDescriptorBase< T>()`, ktorá ako parameter zoberie jeden z descriptorov a vyplní spoločnú časť.

ReportDescriptorInterpreter slúži na interpretovanie HID Report Descriptoru, ktoré prebieha v 2 fázach:

1. `ParseReportDescriptor()` – rozparsuje súvislé dáta reprezentujúce Report Descriptor a vytvorí z nich strom kde vrchol reprezentujeme štruktúrou `ReportDescTreeStruct` (ktorú si detailnejšie opíšme nižšie v sekcii 4.7.1). Tento strom slúži len na jednoduchšiu intepretáciu celého Report Descriptoru. Túto metódu voláme priamo z konštruktora interpreteru.
2. `Interpter()` – slúži na interpretovanie Report Descriptoru. Tu už nepracujeme s dátami poskytnutými QTableWidgetItemom ale so stromom, ktorý sme si vytvorili pomocou `ParseReportDescriptor()` metódy.

SetupPacketInterpreter slúži na interpretovanie Setup Paketu.

StringDescriptorIntepreter slúži na interpretovanie String Descriptoru.

InterruptTransferInterpreter
Slúži na interpretovanie Interrupt Transferu, takže musí zároveň riešiť aký typ zariadenia daný input poslalo. Z hlavičky paketu prenosu vieme zistiť adresu zariadenia, ktoré zariadenie poslalo daný input. V `Interpret()` metóde prejde všetky zariadenia, ktoré máme uložené a nájdeme zariadenie s odpovedajúcou adresou. Zároveň sa musíme pozerať na číslo riadku na ktorom sa nachádza náš QTableWidgetItem z ktorého berieme dáta. Môže sa totiž stať, že pripojíme konkrétne zariadenie na zbernicu (napríklad myš) a USB Host mu nastaví adresu

napríklad 9. Neskôr zariadenie odpojíme a napojíme namiesto neho iné (napríklad klávesnicu) a USB Host mu priradí teraz už voľnú adresu 9. V takomto prípade máme v liste zariadení 2 s rovnakou adresou a nevieme teda, ktoré z nich si máme vybrať. Toto vyriešime tým, že si u každého zariadenia budeme pamätať rozsah riadkov v ktorom boli pakety s touto adresou posielané ním. Ak také nenájdeme, nemáme informácie o danom zariadení a teda nevieme interpretovať jeho input. Ak sa nám podarí dané zariadenie rozpoznať, vyberieme ešte pomocou metódy `GetInputParser` štruktúru, ktorá presnejšie opisuje daný input a následne zavoláme už konkrétny interpreter pre naše zariadenie. Keďže sme sa na začiatku rozhodli, že budeme podporovať 3 HID zaridenia, máme na výber z týchto možností:

1. `MouseInterpreter` – interpretuje zariadenie myši.
2. `KeyboardInterpreter` – interpretuje zariadenie klávesnice. Pre získanie názvu odpovedajúceho tlačidla podľa jeho key codu používame dokumentáciou definované Usage Names [88].
3. `JoystickInterpreter` – interpreturj zariadenie joysticku.
4. `UnknownDeviceInterpreter` – snaží sa o obecnú interpretáciu inputu zariadenia, ktoré nie je ani jedno z vyššie uvedených.

Teraz už poznáme všetky Interpreter triedy nášho programu a musíme ešte vyriešiť ako si AdditionalDataModel vyberie ten správny. Ako sme už spomínali vyššie, budeme to riešiť pomocou návrhového vzoru factory, konkrétne nám na to bude slúžiť trieda InterpretFactory.

4.4.11 InterpretFactory

Trieda pomocou ktorej zvolíme správny interpreter pre AdditionalDataModel. Keďže bude vytvárať inštancie jednotlivých interpreterov, obsahuje nasledujúce dátové položky:

- `TreeItem* rootItem` – koreň stromu.
- `QTableWidgetItem* item` – pointer na item, ktorý v sebe obsahuje dáta.
- `AdditionalDataModel* additionalDataModel` – pointer na model pre zvyškové dáta paketu.
- `HeaderDataType dataType` – typ prenosu/descriptoru, ktorý reprezentujú dáta.

Následne obsahuje metódu `GetInterpreter()` v ktorej pomocou položky *dataType* vráti novú inštanciu správneho interpreteru.

Takže jediné čo AdditionalDataModel musí urobiť keď chce vytvoriť strom pre konkrétny input je vytvoriť si novú inštanciu InterpretFactory, získať správny interpreter a na ňom zavolať metódu `Interpret()`.

Momentálne si povieme niečo ešte o triedach a štruktúrach, ktoré nijako priamo neriadia data-flow aplikácie, ale využívame ich všade naprieč naším programom.

4.5 HIDDevices

HIDDevice je trieda implementujúca logiku súvisiacu s HID zariadeniami. Trieda je implementovaná návrhovým vzorom singleton, pretože obsahuje viaceré dátové položky a funkcie, ktoré by mali byť jednotné a chceme k nim mať prístup z viacerých častí programu. Obsahuje `std::vector< BusDevice>devices` – vector zariadení ktoré boli/sú pripojené na zbernicu počas zachytávania paketov. Zároveň v nej máme implementované veľmi dôležité funkcie `CreateDevice` a `ParseHIDReportDescriptor`

CreateDevice(QByteArray packetData) – táto funkcia sa volá z ItemManagera počas spracovávania paketu v momente, keď máme paket v ktorom zariadenie pošle Configuration Descriptor. Je zodpovedná za vytvorenie nového zariadenia a jeho pridanie do *devices*.

ParseHIDReportDescriptor(QByteArray packetData, USHORT interfaceIndex) – takisto ju voláme z ItemManageru počas spracovania paketu. Je zodpovedná za rozparsovanie HID Report descriptoru. Podľa *interfaceIndex* (číslo interfacu zariadenia s ktorým sa Report Descriptor viaže) si nájdeme konkrétne endpointy, ktorým bude tento Report Descriptor priradený. *packetData* reprezentujú dáta paketu – v nich preskočíme hlavičku paketu a začneme spracovávať dáta, ktoré reprezentujú Report Descriptor. Podľa HID dodatku k USB špecifikácii [18] má každý item hlavičku z ktorej zistíme tag, veľkosť a typ itemu, pričom tieto tri parametre jednoznačne určujú daný item. Dáta tak spacovávame sekvenčne po jednotlivých itemoch a postupne vytvárame jednotlivé HIDReportDescriptorInputParse štruktúry (viac si o nich povieme nižšie v sekcii 4.7.1) reprezentujúce jednotlivé inputy zariadenia.

4.6 DataHolder

Trieda obsahujúca rôzne dátové konštanty a funkcie, ktoré potrebujeme v rôznych častiach programu. Z tohto dôvodu je takisto naimplementovaná ako Singleton. Hlavné položky ktoré obsahuje:

- `TRANSFER_LEFTOVER_DATA`, `TRANSFER_OPTIONAL_HEADER` a `USBPCAP_HEADER_DATA` – konštanty určujúce Qt UserRole pri ukladaní/načítaní jednotlivých QByteArray z QTableWidgetItem pomocou `setData()`/`data()`.
- `BYTES_ON_ROW` – konštanta určujúca počet buniek na jeden riadok v hexdumpe.
- `DataColors` – mapa ktorá priradzuje jednotlivému typu prenosu/descriptoru farebné zložky ktoroými budú jeho dáta zvýraznené v hexdumpe a Color Mape.
- `FillDataColorMap()` – metóda naplňujúca *DataColors* fixnými hodnotami.
- Metódy, ktoré vracajú textovú reprezentáciu rôznych enum konštánt.

4.7 PacketExterStructs.hpp

Je súbor skladajúci sa z dvoch častí:

- USBPCap štruktúry – časť tohto súboru definuje USBPCap štruktúry. Táto časť nie je napísaná autorom tohto projektu, ale je prevzatá zo súboru USBPcap.h [89], ktorý je súčasť open-source zdrojových kódov USBPcapu.
- Nami definované štruktúry a enumy, ktorým sa budeme bližšie venovať v naledujúcej sekcii 4.7.1

4.7.1 ExternStructs

Obsahuje definíciu všetkých štruktúr a enumov, ktoré používame v programe. Teraz is vysvetlíme vzťahy a význam rôznyh štruktúr:

DataTypeColor štruktúra reprezentujúce farebné zložky RGBA.

ReportDescTreeStruct štruktúra slúžiaca na jednoduchšiu interpretáciu Report Descriptoru v stromovej štruktúre. Obsahuje 3 základné parametre hlavičky itemu (tag, veľkosť a typ), QByteArray data reprezentujúci dáta daného itemu, bool premennú určujúcu či sa jedná o koreň stromu a ako posledné vector shared pointerov na svojich synov.

HIDDescriptor štruktúra reprezentuje HID Descriptor definovaný v HID dodatku k USB špecifikácii [90].

InputValues štruktúra reprezentujúca jeden Input item v Report Descriptore. Obsahuje všetky povinné itemy, ktoré v sebe musí Report Descriptor zahrnúť aby opísal dáta posielané zariadením ako napríklad *Report Size*, *Usage Page* atď. Obsahuje takisto niektoré nepovinné itemy (*Usage Minimum*, *Usage Maximum*), ktoré momentálne nepoužívame, ale mohli by sa nám hodiť v prípade rozšírenia našej aplikácie o ďalšie zariadenia.

HIDReportDescriptorInputParse štruktúra reprezentuje jeden konkrétny input report zariadenia. Ako už vieme, zariadenie môže mať k jednému endpointu asociovaných viacero reportov a v takom prípade musia byť prefixované Report ID. V tejto štruktúre si všetky tieto informácie uchovávame a bližšie ich opisujeme v zdrojových kódoch, ktoré sú poskytnuté ako príloha k teto práci.

BusDevice je štruktúra, ktorá reprezentuje jedno konkrétne zariadenie ktoré bolo pripojené na zbernicu počas zachytávania paketov. Najdôležitejšie položky sú adresa zariadenia, ktorú jej pridelil USB host počas konfigurácie a vector HIDReportDescriptorInputParse štruktúr aby sme vedeli interpretovať všetky druhy inputov daného zariadenia.

Týmto sme si prešli celkovú stavbu našej aplikácie a mali by sme mať obecný prehľad akým spôsobom náš program funguje. Všetky detailnejšie informácie o popise jednotlivých položiek, tried alebo metód nájdeme v prílohe tejto práce v

zdrojových kódoch alebo poskytnutej dokumentácie vygenerovanej z dokumentačných komentárov.

5. Možnosti rozšírenia

V tejto kapitole sa pozrieme na rôzne možnosti a smery, akými by sme mohli našu aplikáciu ďalej rozšíriť.

5.1 Ukladanie výstupu do súboru

Niektoré analyzátory ponúkajú ukladanie výpisu do rôznych súborov. V prípade ak by sme sa rozhodli našu aplikáciu rozšíriť o možnosť ukladania analýzy do súboru, museli by sme si rozmyslieť niekoľko nasledujúcich vecí:

Formát výpisu
Akým spôsobom bude vyzerať výstup analýzy. Na výber máme viacero možností od obyčajného textového výstupu až po obrázky alebo rôzne tabuľky. Na začiatok by nebolo príliš zložité zaintegrovať výpis analýzy v textovom formáte. Všetky dáta máme interne reprezentované v podobe QByteArray alebo stromovej štruktúry, ktoré je možné jednoducho prechádzať. Samozrejme musíme myslieť na to, že textový výpis nie je tak flexibilný a interaktívny ako prípadný QTreeView, takže vyobrazenie stromovej štruktúry by pravdepodobne nevyzeralo ideálne.

Rozhodnutie užívateľa, či bude danú analýzu ukladať do súboru alebo nie
Musíme myslieť na to, v akej fáze programu umožníme užívateľovi sa najneskôr rozhodnúť o ukladaní danej analýzy do súboru. Momentálne si všetky dáta uchovávame v samostatných QTableWidgetItemoch uložených v QTableWidgete hlavného okna aplikácie. Užívateľ má ale možnosť toto okno „vyčistiť“ pomocou tlačidla Clear 4.3.1, ktoré vymaže všetky QTableWidgetItemy a tým pádom stratíme referenciu na všetky dáta, ktoré v sebe dané itemy uchovávajú. Doteraz sme sa tým nemuseli zaoberať, pretože dané dáta sem potrebovali na analýzu paketov reprezentovaných riadkom v QTableWidgete, takže akonáhle bol riadok vymazaný pomocu Clear tlačidla, detailnejšia analýza daného paketu už nebola možná. Niektoré možné riešenia tohto problému by mohli byť nasledujúce:

- V prípade, že umožníme užívateľovi si uložiť analýzu do súboru len do momentu pokiaľ nestlačí tlačidlo Start, máme na výber prakticky 2 možnosti:

  1. Ukladať si referencie na jednotlivé QTableWidgetItemy až do momentu skončenia programu, kedy jednorázovo zapíšeme do súboru celú analýzu.
  2. Priebežne zapisovať do súboru analýzu jednotlivých paketov.

  Riešenie 1 prináša tú nevýhodu, že si musíme držať v pamäti všetky dáta paketov a zároveň by ukončenie aplikácie trvalo dlhšiu dobu, pretože by sa musela postupne vykonať a zapísať detailná analýza každého paketu. Naopak riešenie 2 nás zbavuje oboch týchto problémov, ale mohla by mať nepriaznivý dopad na užívateľskú interakciu s aplikáciou, pretože v pozadí by prebiehal proces zapisovania a analýzy. To by sa samozrejme dalo

optimalizovať rôznymi spôsobmi – zapisovať len v momente pokiaľ užívateľ neinteraguje s aplikáciou, využiť metódy paralelného programovania a na zápis/analýzu použiť viac vláken, atď.

- V prípade, že umožníme užívateľovi rozhodnúť o zápise do súboru hocikedy v priebehu používania aplikácie, musíme mať dáta jednotlivých paketov k dispozícii aj po ich odstránení z QTableWidgetu tlačidlom Clear. Ako toho dosiahnuť sme naznačili vyššie v riešení 1 spolu s jeho nevýhodami.

Dôležité je ale spomenúť, že pridaním ukladania výpisu do súboru nijakým vážnym spôsobom nezasahujeme do programu a nemodifikujeme už naimplementované časti. Všetky dáta máme v aplikácii pripravené a jediné čo nám ostáva vyriešiť je ich spracovanie do súboru.

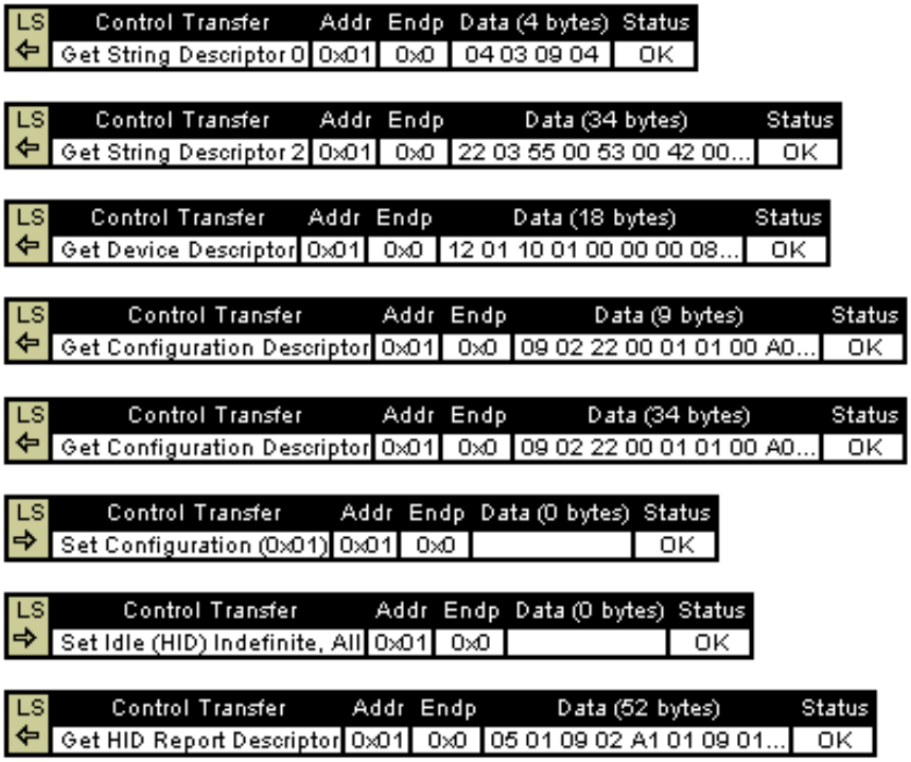
5.2 Iná vizuálna reprezentácia dát

Momentálne vyobrazujeme dáta za pomoci viewerov – QTableView, QTreeView. Ako sme si už spomínali vyššie, celé to prebieha na základe Model/View architektúry, ktorá nám umožnuje od seba oddeliť dáta a spôsob akým ich vyobrazujeme. Na základe toho by pre nás nemalo byť tak náročné implementovať nové spôsoby vizuálnej reprezentácie dát. To by sme vedeli dosiahnuť pridaním nového typu vieweru. Mohli by sme si vybrať z už existujúcich, alebo si kľudne naimplementovať vlastný, ktorý by odpovedal našim predstavám vyobrazenia dát. Momentálne máme vytvorených viacero modelov pre špecifické časti našich dát (HexdumpModel pre všetky dáta paketu na vyobrazenie hexdumpu, USBPCapHeaderModel pre hlavičku paketu na jej vyobrazenie pomocou stromovej štruktúry, atď.). Všetky tieto modely sme implementovali z dôvodu aby sme jednoducho vedeli vyobraziť dáta v jednotlivých vieweroch. Preto by pridanie nového vieweru malo pravdepodobne za následok nutnosť implementovať aj tomu odpovedajúci model.

Zaujímavý spôsob vyobrazenia dát by bolo napríklad pomocou koláčového grafu. Vedeli by sme tak vizuálne zobraziť pomer rôznych dát ako napríklad:

- pomer rôznych typov prenosov (Control/Interrupt/Bulk/Isochronous) počas zachytávania paketov.
- pomer veľkosti dát hlavičky paketu a zvyškových dát.
- pomer veľkosti dát poslaných zariadením a USB hostom.
- pomer zvyškových dát v paketoch vzhľadom na typ prenosu.

Takisto by mohlo byť zaujímavé vyobraziť dáta viac grafickým spôsobom, napríklad ako je ukázané na obrázku 5.1 nižšie.



Obr. 5.1: Ukážka grafického vyobrazenia paketov. Obrázok prevzatý zo stránky USB Made Simple [91]

5.3 Pridávanie nových Interpreterov pre descriptory

Vrámci rozšírenia programu by sme mohli chcieť pridať analýzu pre nové descriptory. V takomto prípade musíme vyriešiť nasledujúce:

- Definícia descriptoru – musíme si rozmyslieť odkiaľ zoženieme definíciu daného descriptoru. Môžeme si ho buď nadefinovať sami alebo použiť stávajúce definície z rôznych knižníc. Momentálne si sami definujeme len jeden descriptor – HID Descriptor a všetky ostatné descriptory ktorých analýzu podporujeme sú definované v súbore usbspec.h.
- Aby sme vedeli v programe tento druh descriptoru rozpoznať, musíme si pridať jeho číselnú reprezentáciu do enumov:
  1. *HeaderDataType* – umožní novú položku v *InterpreterFactory*.
  2. *DesriptorTypes* – umožní rozpoznanie daného descriptoru z údajov Setup Paketu. Táto hodnota sa bude musieť presne zhodovať s hodnotou descriptoru definovanou USB špecifikáciou.
- Definovať v metóde `ItemManager::GetDataType()` prevod z *DescriptorTypes* enumu na enum HeaderDataType.
- Naimplementovať nový Interpreter pre daný descriptor.
- Pridať do *InterpreterFactory* možnosť vytvorenia Interpreteru pre daný descriptor.

5.4 Pridanie Intepreteru pre Interrupt Transfer

O analýzu zariadení, ktoré svoje dáta posielajú cez Interrupt Transfer sa momentálne stará *InterruptTransferInterpreter*, ktorý pôsobí skôr ako ďalšia factory než ako samotný interpreter. Pre pridanie nového zariadenia by sme sa museli postarať o nasledujúce:

- Číselne zareprezentovať dané zariadenie pridaním novej položky do *SupportedDevices* enumu.
- Pridanie tohto zariadenia do našej mapy podporovaných zariadení *deviceMap*.
- Pridanie novej položky v metóde `InterruptTransferInterpreter::Interpret()` v časti kde sa určuje o aké zariadenie sa jedná a na základe toho sa volá daný interpreter.
- Naimplementovanie nového Interpreteru pre dané zariadenie.

5.5 Pridanie analýzy pre Isochronous a Bulk Transfer

Momentálne v aplikácii rozpoznávame Isochronous a Bulk transfer len na úrovni hexdumpu, kde jednotlivé bunky zafarbujeme im odpovedajúcim farbám. Aplikácia ale ponúka dostatočne obecný návrh, ktorý umožnuje možnosť pridania aj sémantickej analýzy práve pre tieto typy prenosov. Typ transferu máme uložený v hlavičke paketu a takisto si ho predávame ako parameter v prípade vytvárania novej inštancie `DataVieweru`. Následne pre interpretovanie zvyškových dát paketu vytvárame novú inštanciu `AdditionalDataModel`, ktorá používa `InterpreterFactory` na výber správneho interpreteru. V tejto factory máme definované položky aj pre Bulk a Isochronous transfer, ktoré ale momentálne vracajú `nullptr`. Stačí nám teda vytvoriť nový Interpreter práve pre tieto transfery a pridať ho do factory. Je celkom pravdepodobné, že by sme pred interpretovaním jednotlivých zariadení iných prenosov museli najprv zozbierať dáta o formáte ich inputu z descriptorov, ktoré posielajú USB hostovi počas konfigurácie. Pripojenie nového zariadenia na zbernicu a vytvorenie jemu odpovedajúcej štruktúry `BusDevice` riešime v `ItemManager::ProcessPacket()` pomocou metódy `CreateDevice()`. Tá momentálne sekvenčne prechádza dáta paketu a v prípade, že sa jedná o HID/Endpoint/Interface descriptor, vytiahne z nich potrebné dáta. V prípade rozšírenia o nový descriptor je nutné len pridať možnosť rozpoznania tohto typu descriptoru a následne jeho konkrétnu analýzu.

5.6 Možnosť rozšírenia na iné platformy

Hlavná podporovaná platforma našej aplikácie je Windows. V priebehu samotného návrhu sme chceli čo najviac obmedziť viazanie sa na jednu špecifickú platformu, pretože už vtedy sme mohli uvažovať nad prípadným neskôrším rozšírením na iné platformy. To malo za následok niektoré naše rozhodnutia, ako

napríklad výber multiplatformového Qt frameworku, alebo manuálne spracovávanie pcap súborov pomocou QFile namiesto použitia Npcap API. S Windowsom nás ale bohužiaľ stále spája zopár vecí, ktoré by sme museli v prípade rozšírenia na iné platformy riešiť inak. Z hľadiska zdrojového kódu to je používanie Windows knižníc. Tie využívame v týchto prípadoch:

- Pri používaní štruktúr základných descriptorov – tie sú definované v súbore usbspec.h
- Štruktúry definované USBPcapom využívajú dátové typy ako napríklad *USHORT*, *USBD_STATUS*, *UINT32*, ktoré sú definované rôznymi Windows knižnicami.

Ďalší problém je s celkovou integráciou nášho programu s USBPcapom, čo je sniffer ktorý funguje iba na Windowse. Celkové spracovanie pcap súborov a formátovanie jednotlivých paketov úzko súvisi s formátom v akom ich USBPcap ukladá.

6. Užívateľská dokumentácia

6.1 Inštalácia

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

6.2 Orientácia v GUI aplikácie

popis k jednotlivým tlačidlám gui

6.3 Používanie aplikácie

ako spustit live/offline capture, a celkovo ako pracovať s aplikáciou(popis funkcií - doubleClick na item => zobrazi 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 požitá na ukážku konkrétneho inputu zariadenia. https://www.amazon.com/Logitech-Spectrum-Shifting-Personalized-Programmable/dp/B019OB663A.

[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] USB 2.0 Specification – USB Data Flow Model. https://www.usb.org/document-library/usb-20-specification. [súbor usb_20.pdf, kap. 5].

[24] USB 2.0 Specification – USB Endpoint. https://www.usb.org/document-library/usb-20-specification. [súbor usb_20.pdf, kap. 5.3.1].

[25] USB 2.0 Specification – USB Pipe. https://www.usb.org/document-library/usb-20-specification. [súbor usb_20.pdf, kap. 5.3.2].

[26] USB 2.0 Specification – USB konfigurácia. https://www.usb.org/document-library/usb-20-specification. [súbor usb_20.pdf, kap. 9.2.3].

[27] USB 2.0 Specification – USB setup paket. https://www.usb.org/document-library/usb-20-specification. [súbor usb_20.pdf, kap. 9.3].

[28] USB 2.0 Specification – USB štandardné device requesty. https://www.usb.org/document-library/usb-20-specification. [súbor usb_20.pdf, kap. 9.4].

[29] USB 2.0 Specification – USB Configuration Descriptor. https://www.usb.org/document-library/usb-20-specification. [súbor usb_20.pdf, kap. 9.6.3].

[30] USB 2.0 Specification – USB Interface Descriptors. https://www.usb.org/document-library/usb-20-specification. [súbor usb_20.pdf, kap. 9.6.5].

[31] USB 2.0 Specification – USB Endpoint Descriptor. https://www.usb.org/document-library/usb-20-specification. [súbor usb_20.pdf, kap. 9.6.6].

[32] MSDN USB – Device node a device stack. https://docs.microsoft.com/en-us/windows-hardware/drivers/gettingstarted/device-nodes-and-device-stacks.

[33] MSDN USB – Plug and Play Manager. https://docs.microsoft.com/en-us/windows-hardware/drivers/install/pnp-manager.

[34] MSDN USB – Device object. https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/wdm/ns-wdm-_device_object.

[35] MSDN USB – HID Architecture. https://docs.microsoft.com/en-us/windows-hardware/drivers/hid/hid-architecture.

[36] MSDN USB – HID Client. https://docs.microsoft.com/en-us/windows-hardware/drivers/hid/hid-architecture#hid-clients.

[37] MSDN USB – Services. https://docs.microsoft.com/en-us/windows/win32/services/services.

[38] MSDN USB – Preparsed Data. https://docs.microsoft.com/en-us/windows-hardware/drivers/hid/preparsed-data.

[39] Komunikácia HID Clienta s HID Class driverom. https://docs.microsoft.com/en-us/windows-hardware/drivers/hid/opening-hid-collections.

[40] HID Application Programming Interface (API). https://docs.microsoft.com/en-us/windows-hardware/drivers/hid/introduction-to-hid-concepts.

[41] 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“].

[42] libusb knižnica. https://libusb.info/.

[43] HIDAPI knižnica. https://github.com/signal11/hidapi.

[44] Moufiltr – filter driver. https://docs.microsoft.com/en-us/samples/microsoft/windows-driver-samples/mouse-input-wdf-filter-driver-moufiltr/.

[45] Kbfiltr – filter driver. https://docs.microsoft.com/en-us/samples/microsoft/windows-driver-samples/keyboard-input-wdf-filter-driver-kbfiltr/.

[46] USBPcap sniffer. https://desowin.org/usbpcap/.

[47] Pcap file format. https://wiki.wireshark.org/Development/LibpcapFileFormat.

[48] Report Descriptor. https://www.usb.org/sites/default/files/hid1_11.pdf. [str. 33].

[49] HID Usage Tables pre USB. https://usb.org/sites/default/files/hut1_2.pdf.

[50] Konkrétny príklad Report Descriptoru myši. https://www.usb.org/sites/default/files/hid1_11.pdf. [str. 71].

[51] Dear ImGui. https://github.com/ocornut/imgui.

[52] SFGui. https://github.com/TankOs/SFGUI.

[53] Qt. https://www.qt.io/.

[54] Qt Model/View architektúra. https://doc.qt.io/qt-5/model-view-programming.html.

[55] Návrhový vzor Model-View-Controller. https://en.wikipedia.org/wiki/Model-view-controller.

[56] Qt QFileSystemModel class. https://doc.qt.io/qt-5/qfilesystemmodel.html.

[57] Qt QTableView class. https://doc.qt.io/qt-5/qtableview.html.

[58] Qt signal a slot. https://doc.qt.io/qt-5/signalsandslots.html.

[59] Npcap. https://nmap.org/npcap/.

[60] Npcap API. https://nmap.org/npcap/guide/wpcap/pcap.html.

[61] C++ Input file stream class – std::ifstream. https://www.cplusplus.com/reference/fstream/ifstream/.

[62] Qt QFile class. https://doc.qt.io/qt-5/qfile.html.

[63] Qt QByteArray class. https://doc.qt.io/qt-5/qbytearray.html.

[64] Qt QVariant class. https://doc.qt.io/qt-5/qvariant.html.

[65] Qt QFileSystemWatcher class. https://doc.qt.io/qt-5/qfilesystemwatcher.html.

[66] Qt QTimer class. https://doc.qt.io/qt-5/qtimer.html.

[67] Qt QListWidget class. https://doc.qt.io/qt-5/qvariant.html.

[68] Qt QListView class. https://doc.qt.io/qt-5/qlistview.html.

[69] Qt QListWidgetItem class. https://doc.qt.io/qt-5/qlistwidgetitem.html.

[70] Qt ItemDataRole enum. https://doc.qt.io/qt-5/qt.html#ItemDataRole-enum.

[71] Qt QTableWidget class. https://doc.qt.io/qt-5/qtablewidget.html.

[72] Qt QTableWidgetItem class. https://doc.qt.io/qt-5/qtablewidgetitem.html.

[73] Qt QAbstractScrollArea class. https://doc.qt.io/archives/qt-5.11/qabstractscrollarea.html.

[74] Qt QTreeView class. https://doc.qt.io/qt-5/qtreeview.html.

[75] Qt QAbstractItemView class. https://doc.qt.io/qt-5/qabstractitemview.html.

[76] Qt Visual Studio Tools. https://marketplace.visualstudio.com/items?itemName=TheQtCompany.QtVisualStudioTools2019.

[77] Compiler Warning C26812. https://docs.microsoft.com/en-us/cpp/code-quality/c26812?view=msvc-160.

[78] Compiler Warning C26451. https://docs.microsoft.com/en-us/cpp/code-quality/c26451?view=msvc-160.

[79] Compiler Warning C26498. https://docs.microsoft.com/en-us/cpp/code-quality/c26498?view=msvc-160.

[80] Compiler Warning C26495. https://docs.microsoft.com/en-us/cpp/code-quality/c26495?view=msvc-160.

[81] Compiler Warning C6011. https://docs.microsoft.com/en-us/cpp/code-quality/c6011?view=msvc-160.

[82] Qt QPushButton class. https://doc.qt.io/qt-5/qpushbutton.html.

[83] Qt Auto-Connect. https://doc.qt.io/qt-5/designer-using-a-ui-file.html#widgets-and-dialogs-with-auto-connect.

[84] Qt QFileDialog class. https://doc.qt.io/qt-5/qfiledialog.html.

[85] Qt QAbstractTableModel class. https://doc.qt.io/qt-5/qabstracttablemodel.html.

[86] Qt QAbstractItemDelegate class. https://doc.qt.io/qt-5/qabstractitemdelegate.html.

[87] Qt Tree Model example. https://doc.qt.io/qt-5/qtwidgets-itemviews-simpletreemodel-example.html.

[88] USB Keyboard Usage Names. https://www.usb.org/document-library/hid-usage-tables-122. [kap. 10].

[89] USBPcap súbor USBPcap.h v github reporzitári. https://github.com/desowin/usbpcap/blob/master/USBPcapDriver/include/USBPcap.h.

[90] USB Device Class Definition for HID – HID Descriptor. https://www.usb.org/document-library/device-class-definition-hid-111. [kap. 6.2.1].

[91] USB Made Simple – príklad grafického vyobrazenia paketov. https://www.usbmadesimple.co.uk/ums_5.htm. [Tretí obrázok na stránke].

Zoznam obrázkov

1.1 USB topológia. Obrázok prevzatý z USB 2.0 špecifikácie [5]. . . . 4
1.2 Ukážka myší, 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 *endpoint descriptoru* 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

2.1 Ukážka device tree. Obrázok prevzatý z Microsoft dokumentácie [32] 19
2.2 Ukážka konkrétnych device nodov. Obrázok prevzatý z Microsoft dokumentácie [32] 19
2.3 Ukážka vytvorenia konkrétnych PDO Pci.sys driverom. Obrázok prevzatý z Microsoft dokumentácie [32] 20
2.4 Ukážka konkrétneho device tree pre PCI zbernicu. Obrázok prevzatý z Microsoft dokumentácie [32] 21
2.5 Ukážka konkrétneho device stacku myši a klávesnice. Obrázok prevzatý z Microsoft dokumentácie [35] 21

3.1 Tabuľka zariadenía ich *Access Mode*. Zariadenia postupne po riadkoch – myš, joystick a klávesnica 23
3.2 Konkrétny príklad Report Descriptoru myši prevzatý z HID Class Specification [50] 26
3.3 Časť Report Descriptoru reprezentujúca jednotlivé kolekcie 27

3.4 Časť Report Descriptoru reprezentujúca tlačidlá myši 27
3.5 Časť Report Descriptoru reprezentujúca osy X a Y pri pohybe s myšou 28
3.6 Časť Report Descriptoru reprezentujúca ukončenie jednotlivých kolekcií 28
3.7 Vyobrazenie základných informácií o pakete pomocou QListWidget 34
3.8 Vyobrazenie základných informácií o pakete pomocou QTableWidget 35
3.9 Vyobrazenie základných informácií o pakete pomocou QTableWidget 36
3.10 Vyobrazenie hexdumpu s farebným oddelením dát podľa významu 37
3.11 Časť Color Map 38
3.12 Vyobrazenie hexdumpu bez farebného oddelenia dát podľa významu 38
3.13 Vyobrazenie stromovej štruktúry 39

4.1 „Select Components“ časť Qt Setupu 41
4.2 Visual Studio možnosť vybratia Qt verzie. 42
4.3 Visual Studio zvolenie adresára ku *qmake.exe*. 42
4.4 Visual Studio obecné nastavenia projektu. 42
4.5 Visual Studio manuálne nastavenie Qt Installation. 43
4.6 Diagram architektúry aplikácie 44
4.7 Tlačidlá v aplikácii. 45

5.1 Ukážka grafického vyobrazenia paketov. Obrázok prevzatý zo stránky USB Made Simple [91] 62

Zoznam tabuliek

Seznam použitých zkratek

Prílohy

.1 První příloha