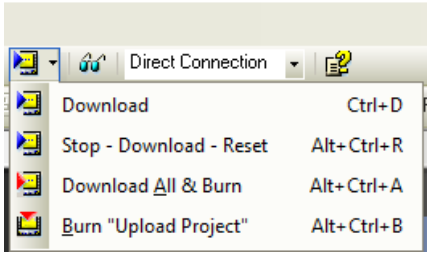
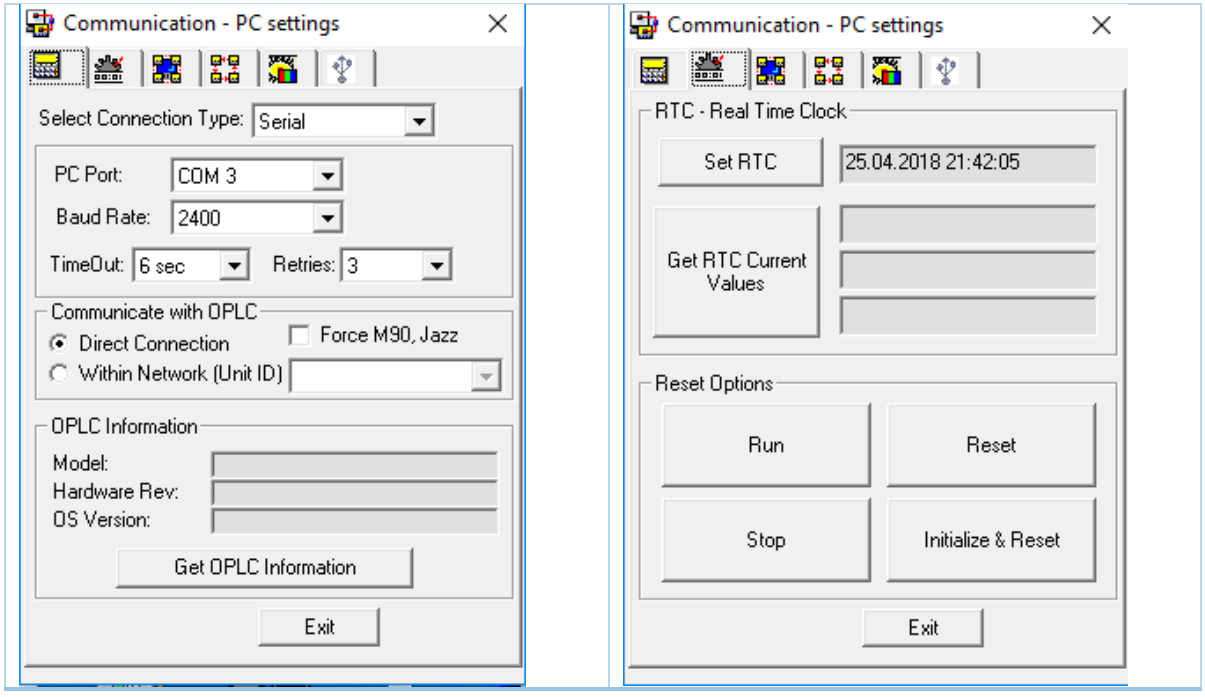
Unitronics visilogic ja UniDownLoader

Unitornics Visilogic on tarkvara keskkond, kus on võimalik koostada programmi tööstusliku kontrolleri programmeerimiseks. Programmeerimine toimub suuresti tähenduslike blokkide abil drag'n drop meetodit kasutades. Lisaks tarkvara laialdasele funktsionaalsusele, mida autor siinkohal välja tooma ei hakka, saab Unitronics Visilogic tarkvara abil ka programmi PLC mällu laadida. Selleks tuleb valida tarkvaras käsklus "Stop – Download - Reset", mis kõigepealt peatab PLC senise programmi, siis laeb PLC programmsesse mällu uue koodi ning lõpuks taaskäivitab PLC koos uue programmiga.



Lisaks sellele, et pidevalt peaks käivitama tarkvara Visilogic ning selle abil programmeerima kontrollerit, saab koodist/projektist teha ka koopia ning seda kasutada siis juba eraldiseisva väiksema tarkvaraga "uniDownloader", kus lisaks tarkvara laadimisele mällu saab teostada lihtsalt veel mitut olulist operatsiooni.



Enamkasutatavamad funktsionaalsused mõlemas tarkvaras kokku on järgmised:

1. GET OPLC – operatsioon, mille abil sisuliselt saab PLC-d identifitseerida
2. RUN – käivitab programmi, kui viimane on peatunud
3. RESET – taaskäivitab kontrolleri
4. STOP – peatab kontrolleri töö
5. STOP – DOWNLOAD –RESET

Nimetatud funktsioonid on lihtsasti käivitatavad Windows operatsioonisüsteemi põhisest kasutajaliidesest, kuid kogu protsessi automatiseerimise huvides on nad kasutamatud, kuna neid operatsioone on raske, kui mitte võimatu kaasata mõne teise programmi töösse. Veelgi enam, taolised operatsioonid võiksid olla kasutatavad ka Linux operatsioonisüsteemis. Näiteks on raske ette kujutada kuidas mõni Java, Python, C vms programm peaks hakkama saama selliste operatsioonide realiseerimisega. See probleem ongi käesoleva töö põhiprobleem ning loetletud operatsioonide tehniline realisatsioon selgub töö käigus.

Kiire lahendus

Winium

Üheks võimaluseks automatiseerida programmi laadimine PLC-sse, on teha seda kasutades olemasolevat kasutajaliidest ning sisuliselt automatiseerida selle füüsiline kasutamine. Veebiarenduses on väga tihti kasutusel vahendid tarkvara testimiseks kõige kõrgemal tasemel ehk kasutaja tasemel. Sobivas programmeerimiskeeles kirjutatakse valmis automaattestid, mis käivitamisel hakkavad kasutama kasutajaliidesega tarkvara nii, nagu seda teeks kasutaja. Testitakse erinevate valikute olemasolu ja funktsionaalsust veebirakenduses. Testitakse, et nuppudele vajutamise puhul toimub ikka see, mis on ette nähtud. Veebirakenduste testimise puhul on kasutusel tarkvara nimega Selenium. Kuna testitakse nähtava kasutajaliidese erinevaid osi, siis on ilmne, et kõik need osad peavad kuidagi olema identifitseeritavad. Seleniumi puhul kasutataksegi näiteks nuppude identifitseerimiseks kas elemendi identifikaatorit, klassikuuluvust, nime, suhtelist teekonda juurelemendist vms.

Analoogiline võimalus on töölaua rakenduste puhul Windows operatsioonisüsteemides. Vabavaralise tarkvarana võib näiteks kasutada Winiumi (Windows + Selenium), mis käitub analoogiliselt Selenium-ile, kuid on kasutatav Windows-i rakendustega. Windows kasutajaliideste elemente inspekteerida on aga tunduvalt keerulisem kui brauseris, kuid see on põhimõtteliselt võimalik. Nimelt on selle jaoks olemas Windows SDK-s spetsiaalne vahend nimega inspect.exe, mis kuvab välja klassikalise töölauarakenduse erinevad objektid ning nende objektidega seotud identifikaatorid (nimi, klass, tüüp jms).

Kasutades eelnimetatud tarkvara on võimalik edukalt automatiseerida paljud inimese poolt tehtavad tegevused kasutajaliideses ning see on ka esimene lahendus, mida autor proovima hakkab. Seda tüüpi automatiseerimine siiski eeldab, et kõikvõimalike seadmete jaoks laboris on eraldiseisvad programmikoodid, mida vahend siis vajadusel PLC-le mällu laadib.

Puudujäägid

Programmi laadimise automatiseerimine kasutajaliidese tasemel omab palju probleeme. Esiteks, lahendus töötaks vaid Windows operatsiooni süsteemil. On vägagi tõenäoline, et server, mis laboris kontrollerite seadistamist haldama hakkab, jooksutab Linux operatsioonisüsteemi. Seega lahend ei sobi, kuna kogu Unitronics tarkvara on loodud vaid Windows operatsioonisüsteemile.

Saadud lahend pole edasi arendatav ning on raskesti hallatav. Nimelt, eeldab lahendus, et programmi täitmise ajal ei toimu serveris parajasti mitte midagi muud. Vastasel korral ei pruugi kasutajaliidese kasutamine enam vastata esialgsele plaanile. Veelgi enam, tulevikus planeeritud kasutajaliidest puudutavad tarkvaralised uuendused teeksid senise lahenduse töö võimatuks.

Kokkuvõtteks, saadud lahendus ei ole piisavalt universaalne, et seda käsitleda, kasutada eraldiseisva tükina, tarkvaralise moodulina.

Alernatiivne lahendus

Selleks, et realiseerida universaalne lahendus, tuleb süveneda olemasoleva süsteemi tehnilisse tausta. On selge, et vajutades kasutajaliideses näiteks Reset nuppu, taaskäivitatakse PLC. Kuna PLC on arvuti külge ühendatud USB kaabli abil, võib oletada, et USB protokolli vahendusel edastatakse kindlaksmääratud sõnum PLC-le. Seega, kui oleks võimalik jälgida detailselt arvuti ja PLC vahelist USB kommunikatsiooni, peaks teoreetiliselt seal kajastuma mingi iseärasus seoses PLC taaskäivitamisega. Samuti peaksid seosed ilmnema ka teiste operatsioonide puhul. Seega on esmaseks eesmärgiks leida sobiv tarkvaraline lahend USB kommunikatsiooni pealtkuulamisele. Esialgse loogika järgi peaks tarkvara võimaldama andmete filtreerimist parameetrite järgi ning tõenäoliselt peaks ka olema võimalus andmeid salvestada/eksportida, et neid töödelda juba järgnevate vahenditega.

Device Monitoring Studio

Device Monitoring Studio on tarkvara mis võimaldab monitoorida, logida, analüüsida USB seadmete vahelist kommunikatsiooni. Tarkvara töötab kõikides Windows versioonides kuid Linuxis mitte. Pealtkuulatud pakette saab ka salvestada ning hiljem uuesti analüüsida. Reaalajas andmevahetuse kuulamisel toimub mustrite avastamine kuni 1 GB andmete ulatuses. Selgusetuks jääb, millises formaadis on võimalik andmeid eksportida/salvestada (csv, xml, json vms). Esialgu oletab autor, et andmevahetus tuleb mahukas ning seega peaks eksisteerima mõni mugav viis saadud info töötlemiseks mõnes tuntud formaadis. Samuti on tegu tasulise tarkvaraga. Tasuta saab kasutada vaid piiratud koguses funktsionaalsust. Üldiselt jääb tarkvara silma selle poolest, et struktureerib selgelt toimuvat andmevahetust. Näiteks sissetulevad paketid värvitakse punaseks ja väljaminevad siniseks jne.

Free USB Analyzer

Tegu on jällegi tasulise tarvaraga. Kasutamine on siiski piiratud natuke erinevalt võrreldes eelmise lahendusega. Nimelt saab päevas teha selle tarkvaraga 5 sessiooni, kusjuures sessioon ei tohi ületada 10 minutit. Sessiooni all peetakse silmas ajavahemikku, mille jooksul toimus andmevahetuse pealtkuulamine. Selgub, et Free USB Analyzer on vaid üks osa suuremast komplektist ning sinna komplekti kuulub veel tarkvaraga, millega on võimalik juba spetsiaalseid protokolle pealt kuulata. Olgu siinkohal nimetatud näiteks Free Serial Analyzer, Free Network Analyzer, Free Virtual COM Analyzer. Autorile tundub, et sellel tarkaral on mida pakkuda: võimalus USB liiklust filtreerida. Arusaamatuks jääb siiski asjaolu, kuidas saadud andmeid salvestada, mis on nende formaat jne. Linux operatsioonisüsteemi tarkvara ei toeta.

Wireshark

Kõikidest teistest andmevahetuse pealkuulamisega tegelevatest tarkvaradest on Wireshark vast kõige kuulsam. Wireshark on tunduvalt laiema haardega tarkvara, kuna võimaldab pealtkuulata mistahes protokolli väga erinevatel kihtidel: USB, TCP, UDP, HTTP jne. Suureks plussiks on filtreerimise olmasolu. Tegemist on täiesti vabavaralise, avatud lähtekoodiga tarkvaraga, mille funktsionaalsus ei jää kuidagi alla ka eespool nimetatud vahenditele. Lisaks toetab tarkvara ka Linux operatsioonisüsteemi. Selgub, et andmeid saab töötlemiseks salvestada mitmes erinevas formaadis. Olgu nimetatud xml, C päise laiendusega fail (.h), json, csv, kui ka tavaline .txt laiendusega fail.

Riistvaraline alternatiiv

Lisaks nimetatud valikutele eksisteerib ka võimalus sättida riistvaraline kontroller või spetsiaalne vahend arvuti ja seadme vahele. See vahend oleks USB signaali füüsiline vahendaja(USB protokolli analüsaator) ning kuvaks vajadusel kõik paketid. Üheks selliseks seadmeks on näiteks Beagle USB analüsaator. Esialgu autor seda teed ei lähe, kuna see eeldab kas mikrokontrolleri programmeerimist liikluse kuulamiseks või siis spetsiaalse riistvaralise vahendi ostmist.

Sobivaks lahendiks arvab autor olevat Wireshark, kuna autoril on sellega varasemaid kohemusi ning kuna viimane täidab analüüsi alguses nimetatud nõudmisi kõige rohkem. Kui peaks tekkima probleeme Wireshark tarkvara kasutamisega on olemas selgelt 2 tarkvaralist ja 1 riistvaraline vahend, mille kasutamise üle minna.

USB protokoll

USB (Universal Serial Bus) protokoll on olemuselt suhteliselt keeruline. Siiski on tegu nii mõneski mõttes kihilise protokolliga ning lõpptulemusena piisab vaid sobiva kihi analüüsimisest, jälgimisest. See lihtsustab arusaamist protokollist. Algatuseks peab autor aga oluliseks USB protokolli põhimõistete tutvustamist, kuna neid on palju ning need on kohati mittetriviaalsed.

Füüsilisel tasemel on USB protokolli puhul tegu tähttopoloogilise võrguga. See tähendab, et on üks juhtseade (näiteks arvuti), mis juhib andmevahetust ning lõplik hulk teisi seadmeid mis on kõik ühendatud keskse seadme (juhtseade) külge. Kuna juhtseade kontrollib andmevahetust, siis on viimane alati andmevahetuse alustaja. Kuigi andmevahetust juhtseadme ja teiste seadmete vahel nimetatakse kahesuunaliseks, siis sisuliselt ei ole need seadmed "iseteadlikud" ning vastavad lihtsalt neile saadetud päringutele. Sisuliselt taandub kahesuunaline andmevahetus kas seadmest lugemiseks või seadmesse kirjutamiseks. Mõlemat teostab juhtseade. Võib tekkida õigustatud küsimus, et kuidas saab näiteks arvutihiir töötada, kui andmevahetust alustab ainult arvuti. Arvuti ei saa põhimõtteliselt teada, millal on õige aeg andmevahetuse alustamiseks. Nii hiir kui ka klaviatuur kuuluvad sellisesse seadmete klassi nagu HID. See tähendab, et juhtseade käib teatud intervalliga pidevalt pärimas(lugemas) näiteks hiire olekut ja kui see muutub, siis saab juhtseade selle teada.

Loogilisel tasemel on USB protokolli puhul tegu siinitopoloogilise võrguga. See tuleneb sellest, et loogilisel tasemel saadab juhtarvuti päringu kõikidele võrgus olevatele seadmetele, kuid vastab vaid see, kellele see adresseeritud oli. Seega kõigil seadmetel on justkui kasutada sama loogiline siin.

Igal USB seadmel on mingi arv endpointe. Endpoint on justkui selle seadme funktsionaalne allsüsteem, mille vastutusalasse kuuluvad kindlad ülesanded. Endpointide arv võib varieeruda olenevalt seadme funktsionaalsuste varieeruvusest, kuid keskmisel USB seadmel ei ole üle kahe endpointi. Lisaks kehtib reegel, et igal USB seadmel peab olema vähemalt üks EP.

Igal Endpointi juurde kuulub kaks puhvrit. Viimaseid nimetatakse IN ja OUT puhvriteks. Kui seade tahab juhtseadmele midagi saata, siis kirjutab ta oma andmed oma Endpointi IN puhvrisse. Juhtseade loeb need andmed USB seadme IN puhvrist. Nagu näha, siis juhtseade peab teadma, millal andmeid lugeda. Kui juhtseade tahab saata andmeid USB seadmele, siis kirjutab ta oma andmed USB seadme OUT puhvrisse. Sealt edasi loeb USB seade puhvrist need andmed. Siit selgub, et kogu tarkvaraline tarkus, mis seisneb USB andmevahetuse taga, asub juhtseadmes, milleks tihti on arvuti. See on ka

peamine põhjus miks igal seadmel peab olema oma kindlaksmääratud USB draiver, sest just viimane on see, mis omab kogu informatsiooni, mida on vaja andmete mõlemasuunaliseks vahetamiseks.

Eespool nimetatud lugemis- ja kirjutamisoperatsioone nimetatakse üldistatult transaktsioonideks. Iga transaktsioon koosneb kolmest paketist: Token pakett, Data pakett, Handshake pakett. Iga transaktsioon ei pea sisaldama sisulisi andmeid, seega võib Data pakett olla ka tühi. Token pakett määrab ära selle, millisele USB seadmele ning millisele endpointile on transaktsioon mõeldud. Üldiselt võib Token paketti kujutada järgmiselt: [address][endpoint][direction]. Data pakett kannab endas sisulisi andmeid ning Handshake pakett kinnitab transaktsiooni. USB standardi kohaselt on defineeritud neli eri tüüpi transaktsiooni: Control transfer, Bulk transfer, Interrupt transfer ja Isochronous transfer. Need on ühtlasi ka elemendid, mida Wireshark välja kuvab ning neid autor edaspidi ka analüüsima hakkab.

Control Transfer

Juhtseade teostab seda sorti transaktsioone, kui on vaja informatsiooni USB seadme kohta või tema seisundite kohta. Lisaks kui eesmärgiks on muuta või teada saada USB parameetreid, siis seda tehakse nende transaktsioonide abil. USB standardi järgi peab igal USB seadmel olema vähemalt üks endpoint. Seda kutsutakse EP0. EP0 on funktsionaalne allsüsteem, mis vastutab eelpool nimetatud küsimuste eest. Kõik kontrolltransaktsioonid tehakse sellele endpointile ning kui seda ei oleks, poleks võimalik isegi teada saada, mis seadmega on tegu. Selleks, et teada saada, mis seadmega on tegu ja kuidas sellega suhtlema peaks (piirangud jne), on ette nähtud standardsed kontrolltransaktsioonid ning iga USB seade peab olema võimeline neile vastama. Need on transaktsioonid, mille põhjal arvuti tuvastab, millist draiverit kasutada või kui seda pole, siis millist draiverit võrgust allalaadimiseks otsida. See on ainuke viis, kuidas arvuti teeb põhimõtteliselt vahet HP prinetril või Logitech arvutihiirel.

Bulk Transfer

Bulk transaktsioonid on suuremahuliste andmete vahetuseks seadmete vahel. Nende transaktsioonide puhul rakendatakse veakontrolli (CRC) ning selliste transaktsioonide peale on tihti ehitatud mõni kõrgetasemeline protokoll. Põhiline erinevus kontroll transaktsioonidega võrreldes on see, et neid transaktsioone ei tehta EP0 endpointi, vaid teistesse võimalikesse endpointidesse. Mass Storage seadmed on klassikaline näide USB seadmetest, kus Bulk transaktsioone kasutatakse.

Interrupt transfer

Interrupt transaktsioonid on mitteperioodiliste teadete andmiseks. Näiteks arvutihiir ning kõik seadmed, mis kuuluvad HID klassi, on seadmed, mille puhul kasutatakse interrupt transaktsioone. Selle põhiliseks tunnuseks on see, et juhtseade pidevalt kontrollib, kas USB seadme olek on muutunud vms.

Kui USB seade ühendada juhtseadme(arvuti) külge, siis esimese asjana omistatakse seadmele aadress ning siis teostab arvuti kontrollpäringud, millega ta teeb USB seadme erinevad parameetrid kindlaks. Lisaks paljudele parameetritele, mis seadet kirjeldavad ning mida autor siinkohal välja tooma ei hakka, kuuluvad parameetrite hulka ka tootja identifikaator, seadme identifikaator, maksimaalne paketi suurus, klass kuhu seade kuulub jne.

USB andmevahetuse jälgimine Wiresharkis

Et teada saada, milline informatsioon liigub arvuti ja PLC vahel erinevate operatsioonide korral, kasutab autor tarkvara nimega Wireshark. Põhilised küsimused millele autor vastuseid otsib on järgmised:

1. Kas iga kord, kui sama operatsioon kindlate parameetritega teostatakse, on andmevahetus sama?
2. Kui kindlal operatsioonil parameetreid muuta, siis mis muutub andmevahetuses?
3. Kui palju transaktsioone toimub operatsiooni korral?
4. Mis toimub eri operatsioonide puhul? Kas nendel on mõni üldine ühisosa?
5. Kuidas võrrelda kahte erinevat andmevahetust?

Lihtsuse huvides alustab autor sellise operatsiooniga nagu GET OPLC, mille abil UniDownloader kasutajaliideses kuvati välja mõningased andmed PLC kohta. Kuna Wireshark kuvab kõikide seadmete USB andmevahetuse samaaegselt, siis esimese asjana on vaja filtreerida vaid need transaktsioonid, mis on seotud PLC-ga. PLC aadressi saab autor teada jälgides andmevahetust hetkel, kui seade ühendadakse arvutiga. Kui saadud aadress oleks näiteks 4, siis esimeseks filtriks olekski:

usb.device_address == 4

Järgmine asi, mida vaadata on transaktsioonide arv. Seda numbrit saab korduskatsetel võrrelda ning see annab aimu, kas andmevahetus on iga kord sama. GET OPLC puhul on kolme katse tulemusena transaktsioonide arv vastavalt 331, 334, 331. Lisaks sellele avastab autor kiiresti, et eksisteerib väike hulk Bulk transaktsioone ning keskmiselt on neid alla 2% kõikidest transaktsioonidest ning need esinevad tihedalt järjestatuna andmevahetuse lõpupoole. Autorile saab kiiresti selgeks, et need transaktsioonid on olulised, kuna samade operatsioonide korral on need alati konstantsed, kuid erinevate operatsioonide puhul on alati teatud erinevus. Bulk transaktsioone filtreerin kõikidest transaktsioonidest järgnevalt:

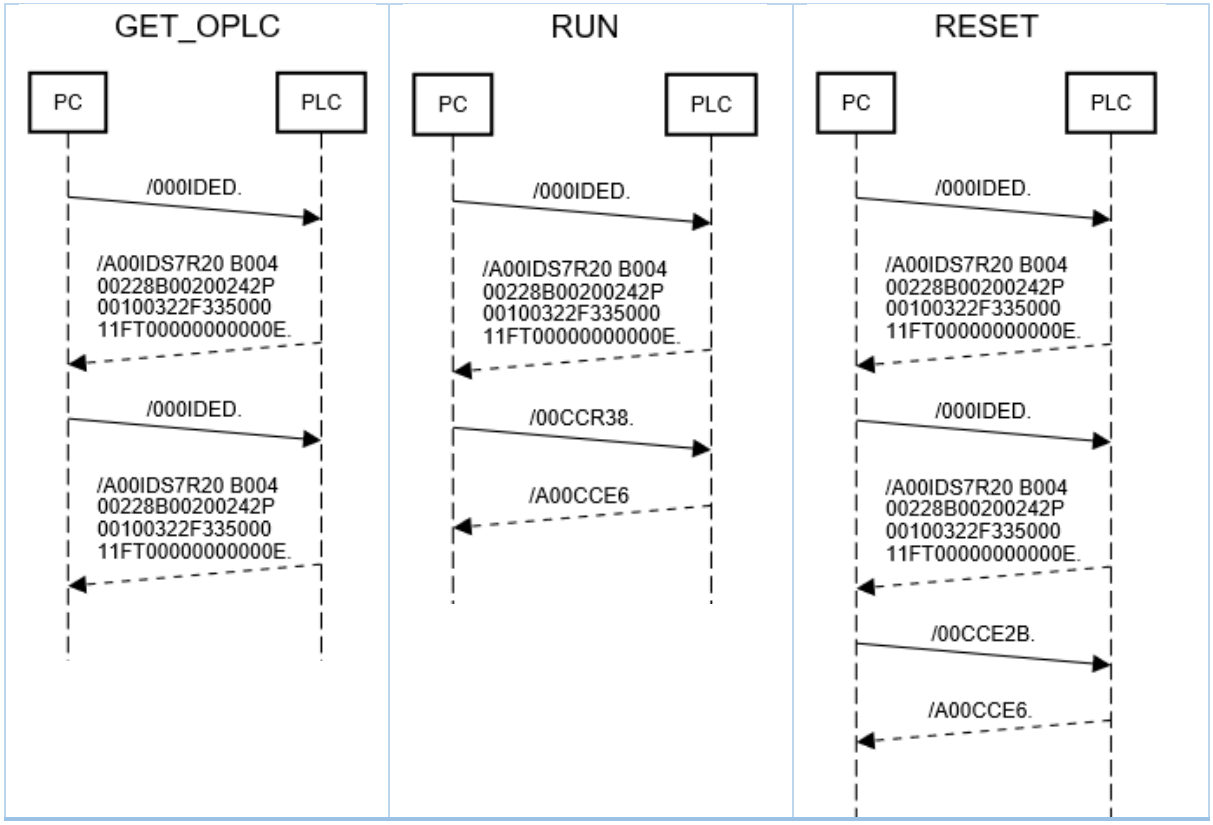
usb.device_address == 4 && usb.transfer_type == 0x03

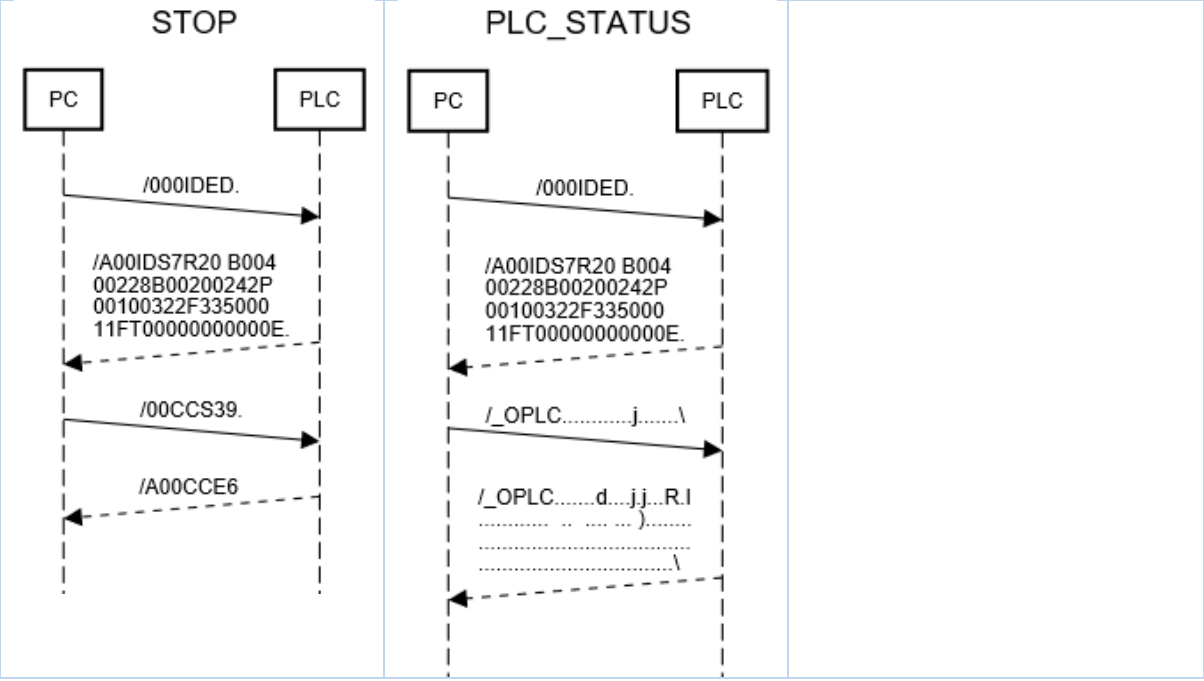
Arvule 0x03 vastab siin Bulk transaktsioon. Samas 0x02 on näiteks Control transaktsisoon.

Autor teeb eelduse, et kuna Bulk transaktsioonid on sisuliste andmete edastamiseks mõeldud transaktsioonitüüp, siis tasub just Bulk transaktsioone erinevate operatsioonide puhul võrrelda. Samas on autor veendunud, et ilma Control transaktsioonide toimumiseta operatsioone läbi viia ei ole võimalik, kuna toimub teatud seadistamine, mille tähendusest autor veel aru ei saa. Edaspidi võetakse aga lahenduse implementeerimisel Control transaktsioonide olemasolu arvesse. Allolevas tabelis on iga operatsiooni kohta teostatud mõõtmisi:

Operatsioon	**Transaktsioonide arv kokku**	**Bulk transaktsioonide arv**
GET_OPLC	331, 334, 331	4
RESET	341, 341, 341	6
RUN	334, 334, 331	4
STOP	334, 331, 334	4
PLC_STATUS	338	4
STOP-DOWNLOAD-RESET	2179	112

Tabelist selgub, et kõikide lihtsamate operatsioonide puhul on transaktsioonide arv suhteliselt väike, ning sisulisi andmeid vahetatakse suhteliselt vähe. Samas viimane operatsioon, mille sisuks on PLC-sse laadida programm, mis seadistab kontrollerile nime ja staatilise IP-aadressi, on tunduvalt mahukam, sisaldades kokku 2179 transaktsiooni, millest 112 on sisulised andmed. Autor arvab, et 112 transaktsiooni kulubki reaalse masinkoodi ülekandmiseks kontrollerisse. Lihtsuse huvides üritab autor kõigepealt analüüsida ja implementeerida lahendus lihtsamatele operatsioonidele ning alles hiljem pöörduda tagasi programmi laadimise juurde. Allpool on kujutatud kõikide lihtsamate operatsioonide interaktsioonidiagrammid, kus toimub sisuliste andmete vahetamine arvuti ja PLC vahel. Andmed on kodeeritud ASCII väärtusteks, kuna ka Wireshark presenteerib neid nii ning ühtlasi on selline kirjapilt rohkem loetavam, kui baidijada. Kuna igale võimalikule baidi väärtusele ei vasta tingimata ASCII sümbol, siis need on asendatud punktiga.





Järeldused

GET OPLC operatsioon teeb sisuliselt kaks identset päringut PLC-le ning saab samuti kaks identset vastust. Kuna on teada, et sellest vastusest peaks saama välja lugeda erinevad PLC-d identifitseerivaid andmeid, siis see selgitab ka seda, miks kõikvõimalike operatsioonide alguses viiakse tegelikult sama protseduur läbi. Üldistatult võib öelda, et iga operatsioon koosneb kahest faasist: seadme identifitseerimine ja siis juba spetsiaalse käsu saatmine ning vastuse saamine. Diagrammidelt on näha, et peale identifitseerimist on kõik käsud vastavalt operatsioonile erinevad ning seega eeldab autor, et just need käsud realiseerivad reaalse operatsiooni. Lisaks on RUN, RESET ja STOP operatsioonide puhul lõpus ka sarnane päring USB seadmele. Kuigi seda ei saa kindlalt väita arvab autor, et tegu on PLC poolse kinnitusega, et operatsioon on edukalt lõppenud.

USB andmevahetuse taasesitamine

Andmevahetus, mis toimub arvuti ja PLC vahel on korraldatud vastava draiveri poolt. See draiver on installeeritud Windows operatsioonisüsteemi ning omab teadmist sellest, milline peab olema andmevahetus vastavate operatsioonide korral. Võib oletada, et kui tegu on mõne spetsiifilise draiveriga, siis ei ole see kasutatav mitte millegi muu kui Unitronics tarkvara poolt. Seega on tarvis andmevahetus pöördprojekteerida ning selleks läheb vaja vahendit, millega saaks mugavalt teha erinevaid USB transaktsioone. Eesmärk oleks taasesitada Wireshark-is nähtud andmevahetus ning loota, et see töötab.

Pikemal uurimisel selgub, et taolisi vahendeid, millega usb transaktsioone edukalt teha, peaks leiduma igas programmeerimiskeeles. Javas oleks võimalik kasutada teegina Java4Usb-d, pythonis PyUSB-d jne. Kõikide selliste teekide ühisosa seisneb selles, et nad põhinevad omakorda C programmeerimiskeele teegil nimega libusb. Windows operatioonisüsteemis tekkis probleem kõikide võimalike ülalnimetatud teekidega. Linux operatsioonisüsteemis ei tekkinud esialgsel katsetamisel ühtegi probleemi ning seega otsustas autor esialgu võttagi kasutusele Linux-i USB andmevahetuse

taasesitaamiseks. Kuna libusb teek oli piisavalt arusaadav, siis ei pidanud autor vajalikuks kasutada Linux-is testimiseks libusb pealisehitusi nagu pyUSB või Java4Usb. Olulised libusb funktsionaalsused, mida autor edaspidi kasutama hakkab on esitatud tabelis.

Funktsioon	**Kirjeldus**
libusb_open_device_with_vid_pid()	Tagastab viite objektile, mida on võimalik hiljem kasutada teostamaks transaktsioone. Identifitseerib objekti tootja id (VID) ja toote id (PID) järgi.
libusb_control_transfer()	Võimaldab teha control transaktsioone.
libusb_bulk_transfer()	Võimaldab teha bulk transaktsioone.
libusb_interrupt_transfer()	Võimaldab teha interrupt transaktsioone.
libusb_kernel_driver_active()	Kontrollib, kas seadmele on kerneli poolt vastavusse juba seotud mõni draiver.
libusb_detach_kernel_driver()	Eemaldab kerneli poolt automaatselt antud draiveri. See on vajalik, sest samal ajal ei saa olla ühel seadmel mitu draiverit.
libusb_attach_kernel_driver()	Seadmele lisatakse automaatne kerneli draiver.

Järgmine küsimus seisneb selles, kuidas teisendada Wireshark-ist saadud transaktsioonid sobivale kujule. Wireshark pakub iseenesest filtreeritud transaktsioonide eksportimiseks csv, txt, json formaati. Lisaks klassikalistele formaatidele on võimalik transaktsioonid teisendada ka C programmeerimisekeele massiivideks. Teisenduse tulemusena oleks vaja baidijada, mis iseloomustab igat transaktsiooni. Kuna viimane viis järjestab transaktsiooni baidid kenasti baidijadana, siis autor otsustab kasutada seda. Sellised baidijadad sisaldavad kogu informatsiooni, mis transaktsiooni puudutas. USB seadme aadress, endpointi number, sisuliste andmete hulk, transaktsiooni suund ning samuti ka sisulised andmed ise, kui neid oli. Baidijada 23. Bait määran ära transaktsiooni tüübi. Nii on joonisel esimese transaktsioonina kujutatud control transaktsiooni. Samas teise puhul on tegu bulk transaktsiooniga ning tema välju tuleb teisiti tõlgendada.

```
static const unsigned char pkt68[36] = {
	0x1c, 0x00, 0xf0, 0xda, 0xce, 0xb5, 0x0d, 0x85,
	0xff, 0xff, 0x00, 0x00, 0x00, 0x00, 0x08, 0x00,
	0x00, 0x01, 0x00, 0x03, 0x00, 0x00, 0x02, 0x08,
	0x00, 0x00, 0x00, 0x00, 0x41, 0x00, 0xff, 0xff,
	0x00, 0x00, 0x00, 0x00
};
static const unsigned char pkt367[35] = {
	0x1b, 0x00, 0x10, 0x00, 0x3d, 0xb5, 0x0d, 0x85,
	0xff, 0xff, 0x00, 0x00, 0x00, 0x00, 0x09, 0x00,
	0x00, 0x01, 0x00, 0x03, 0x00, 0x01, 0x03, 0x08,
	0x00, 0x00, 0x00, 0x2f, 0x30, 0x30, 0x49, 0x44,
	0x45, 0x44, 0x0d
};
```

Control transktsioone tuleb tõlgendada vastavalt allolevale tabelile:

Endpointi number	0x00	22. bait
bmRequestType	0x41	29. bait
bRequest	0x00	30. bait
wValue	0xffff	32, 31 baidid
wIndex	0x0000	34, 33 baidid
wLength	0x00	35. bait
Transaktsiooni tüüp	0x02	23. bait

Bulk transaktsioone tuleb tõlgendada vastavalt allolevale tabelile:

Endpointi number	0x01	22. bait
Transaktsiooni tüüp	0x03	23. bait
Andmete hulk (baitides)	0x08 = 8 baiti	24. bait
Andmed	Viimased 8 baiti	-

Esimese kitsendusena plaanib autor vaadata vaid transaktsioone, mille suund on arvuti poolt USB seadme poole. Selleks tuleb enne andmete baidijadaks teisendamist sisestada Wireshark-is filter, mis transaktsiooni allikaks paneb arvuti. Interaktsioonidiagrammist järeldus, et RESET operatsiooni korral ei ole sisuliselt oluline mida USB seade vastab, sest käskluse saadab niikuinii arvuti ja mingit valideerimist hiljem ei ole. Seega implementeerib autor alguses naiivse lahenduse, kus kõik transaktsioonid teostatakse arvuti poolt USB seadme poole. Kuna baidijada tähenduses on jõudnud autor selgusele, siis järgmiseks sammuks on baidijada konverteerimine libusb teegi poolt käivitatavateks transaktsioonideks. Sisuliselt on autori kavatsuseks automaatselt genereerida kompileeruv C kood, mis taasesitaks baidijada, mille autor varem filtreeris.

Baidijada konverteerimiseks on kõige mugavam kirjutada skript, mis konverteerimise ellu viib. Selleks kirjutab autor Python programmi, mis igast transaktsiooni kujutavast baidijadast loeb välja olulise informatsiooni ning genereerib vastavat tüüpi transaktsiooni koos argumentidega. Sellise programmi käivitamisel konverteeritakse eelpool nimetatud baidijadad järgmisteks C programmeerimiskeele ridadeks:

```
libusb_control_transfer(dev_handle, 0x41, 0x00, 0xffff, 0x0000, in_buffer, 0x00, 1000);
unsigned char out_buffer_0[8] = {0x2f, 0x30, 0x30, 0x49, 0x44, 0x45, 0x44, 0x0d};
libusb_bulk_transfer(dev_handle, 1, out_buffer_0, 8, &actual, 1000);
```

Nagu näha, on tulemuseks üks kontroll transaktsioon ning üks bulk transaktsioon, kus sisuliste andmete maht on 8 baiti. Seega eksisteerib lahendus andmevahetuse taasesitamiseks. Kuna igale transaktsioonile vastab nüüd sisuliselt üks rida, tekivad teatavad võimalused vastata vastamata küsimustele, mis autor varem tõstatas. Seda, kuidas erinevad operatsioonid omavahel ja kuidas leida erinevused erinevatel aegadel tehtud operatsioonide vahel, autor järgnevalt uurima asubki.

Kahe teksti sarnasus

Some tekst here how two texts/operations are comared to each other.

Testimine

Võib tekkida õigustatud küsimus selle osas, kas ka lahend reaalselt töötab. Kuigi on loodud protseduur, mis teoreetiliselt peaks töötama, tuleb läbi viia ka mõned katsed kontrollimaks, kas lahendus on piisav probleemide lahendamiseks. Kuna protseduur on suhteliselt üldine, ei pea selle kasutamine olema seotud just Unitronics PLC andmevahetusega. Just seepärast viib autor esmalt katsed läbi kahel erineval seadme. Nendeks on Arduino UNO arendusplaat ning HP printer.

Arduino UNO arendusplaadi jaoks kirjutab autor valmis programmi, programmeerib sellega arendusplaadi ning salvestab samaaegselt andmevahetuse arvuti ja Arduino vahel. Saadud andmevahetuse konverteerib autor C koodiks. Üllatusena toimub mitmesaja rea kompileerimisel ja käivitamisel tõesti arendusplaadi programmeerimine ning selle tõestuseks hakkab LED tuli vilkuma.

HP printeri jaoks on protseduur analoogiline. Transaktsioonide hulk on tunduvalt suurem Arduino arendusplaadi omast ning ka siin tasub katsetamine ennast edukalt ära. Printer alustab printimist ja printib täpselt selle, mida vaja.

Unitronics PLC erinevate operatsioonide katsetamisel analoogilise meetodiga tekib aga tagasilöök. Autori jaoks seletamatul põhjusel ei reageeri PLC näiteks RESET operatsioonile kuidagi. On selge, et lahenduses on midagi, millega autor veel arvestanud pole. Samas tundub toimunu äärmiselt veider, kuna sisulist erinevust Ardino UNO programmeerimise ja PLC programmeerimise vahel ei ole.