



Elektrotehnički fakultet
Univerzitet u Banjoj Luci

IZVJEŠTAJ PROJEKTOG ZADATAKA

iz predmeta

POUZDANOST ELEKTRONSKIH SISTEMA

Student: Vidić Luka 2201/24

Mentor: Prof. Dr Ivanović Željko

Sadržaj

1. Uvod.....	3
2. Ciljna platforma i hardverska struktura sistema	4
2.1. NUC980 serija mikroprocesora.....	4
3. Priprema operativnog sistema za ciljnu platformu	9
3.1. <i>Buildroot</i>	9
3.2. Konfiguracija Linux jezgra, Linux rukovaoci uređaja, struktura stabla uređaja.....	11
3.2.1. Rukovaoci uređaja (<i>Linux Device Drivers</i>).....	11
3.2.2. Stablo uređaja (<i>Device Tree</i>).....	12
3.2.3. Konfiguracija Linux jezgra za rad u realnom vremenu	16
4. Razvoj aplikacije projektnog zadatka	18
4.1. Protokol IEC 60870-5-104 (IEC 104).....	18
4.1.1. IEC 104 biblioteka – <i>lib60870</i>	22
4.2. Protokol MODBUS RTU	23
4.2.1. MODBUS biblioteka – <i>libmodbus</i>	26
4.3. Mapiranje između IEC 104 i MODBUS protokola.....	26
4.4. Konfiguracija <i>gateway</i> uređaja	28
4.5 Softverska implementacija sistema	31
4.5.1. Implementacija MODBUS mastera.....	32
4.5.2. Integracija MODBUS mastera i implementacija aplikacije gejtveja	36
5. Testiranje funkcionalnosti <i>gateway</i> uređaja.....	43
6. Zaključak i predlozi poboljšanja	48
Literatura.....	49

1. Uvod

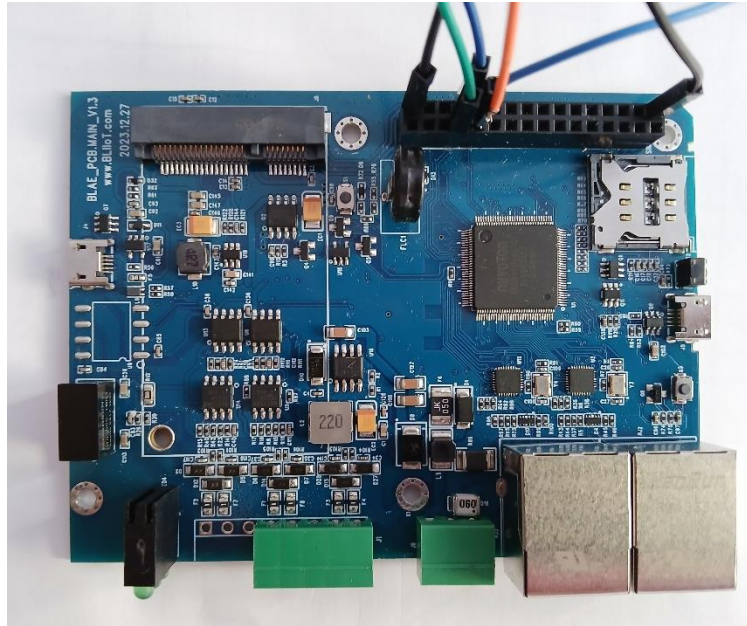
Cilj ovog projektnog zadatka je implementacija *gateway* uređaja između industrijskog protokola IEC 60870-5-104 (nadalje u dokumentu IEC 104) i industrijskog protokola MODBUS RTU. Uređaj je implementiran na komercijalno dostupnoj ploči koja je bazirana na NUC980DK61YC mikroprocesoru proizvođača Nuvoton. Svrha uređaja jeste mapiranje podataka, komandi i poruka IEC 104 protokola na protokol MODBUS RTU tako da se sa strane IEC 104 *master* uređaja (npr. računar sa SCADA sistemom) omogući razmjena podataka i slanje komandi sa MODBUS *slave* uređajima. Sam uređaj spada u široku grupu ugrađenih računarskih sistema te se implementacija sastoji od svih standardnih koraka potrebnih za implementaciju bilo koje aplikacije na ugrađenom sistemu:

- **Izučavanje hardvera** – Prvi i osnovni korak jeste upoznavanje sa hardverom sistema sa kojim se radi. U ovom slučaju razvojna ploča je komercijalna stoga dokumentacija o istoj nije javno dostupna već se izučavanje svodi na izučavanje pojedinačnih komponenti od značaja kao i odgovarajućih postupaka reverznog inženjeringa.
- **Priprema operativnog sistema na razvojnoj ploči** – Razvoj aplikacija na ugrađenim računarskim sistemima uglavnom se svodi na razvoj za ciljni operativni sistem koji će se izvršavati na razvojnoj ploči čime se omogućava portabilnost aplikacija na različite razvojne ploče. Najčešće korišćeni su Linux bazirani operativni sistemi ili neki jednostavniji RTOS operativni sistemi. U nekim slučajevima ovaj korak se može preskočiti, kada je u pitanju razvoj *bare-metal* aplikacija. U svrhu izrade ovog projekta korišćen je Linux operativni sistem.
- **Implementacija aplikacije** – Kada se okruženje pripremi za rad nastupa se sa izradom aplikacije. Izbor postojećih biblioteka, implementacija traženih funkcionalnosti te verifikacija istih su osnovni koraci ka implementaciji zahtjevanoj aplikaciji. Za implementaciju ovog projekta glavne korištene biblioteke su *lib60870* i *libmodbus*.

Kroz nastavak dokumenta biće detaljno opisani svi prethodno navedeni koraci i pojedinačni postupci sprovedeni u datim koracima.

2. Ciljna platforma i hardverska struktura sistema

Hardverska implementacija sistema se zasniva na već izgrađenoj i komercijalno dostupnoj štampanoj ploči koja je izvađena iz kućišta postojećeg uređaja (*gateway* slične namjene kao i u ovom projektu) proizvođača BLIIoT [1]. Data ploča prikazana je na slici 2.1.

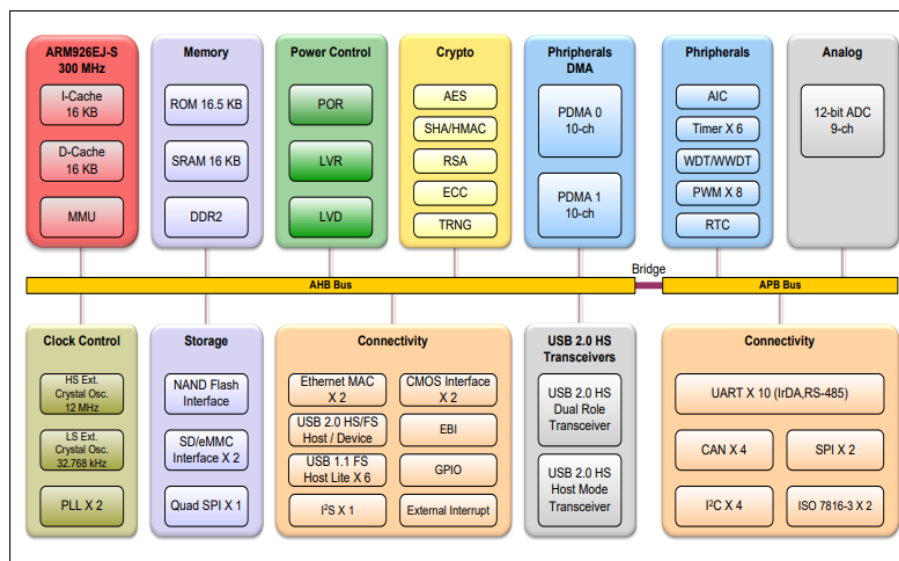


Slika 2.1 – Štampana ploča ciljne platforme

Data platforma bazirana je na mikroprocesoru NUC980DK61YC proizvođača Nuvoton koji predstavlja glavnu upravljačku jedinicu koja izvršava aplikativni program i upravlja periferijama od značaja. Platforma dolazi sa 1 GB fleš (*flash*) memorije proizvođača *Winbond*, dva *Ethernet* porta koji su neophodni za IEC 104 komunikaciju te šest serijskih RS-485 kanala koji se koriste u svrhu MODBUS RTU komunikacije. Pored ovih glavnih karakteristika, ciljna platforma sadrži i dodatne kao što su interfejs za SIM karticu, dodatni serijski USB port u svrhe debugovanja, tastere za resetovanje i više LED za signalizaciju. U nastavku biće detaljnije prezentovana serija mikroprocesora NUC980 s obzirom na centralnu ulogu tog mikroprocesora u ovom sistemu.

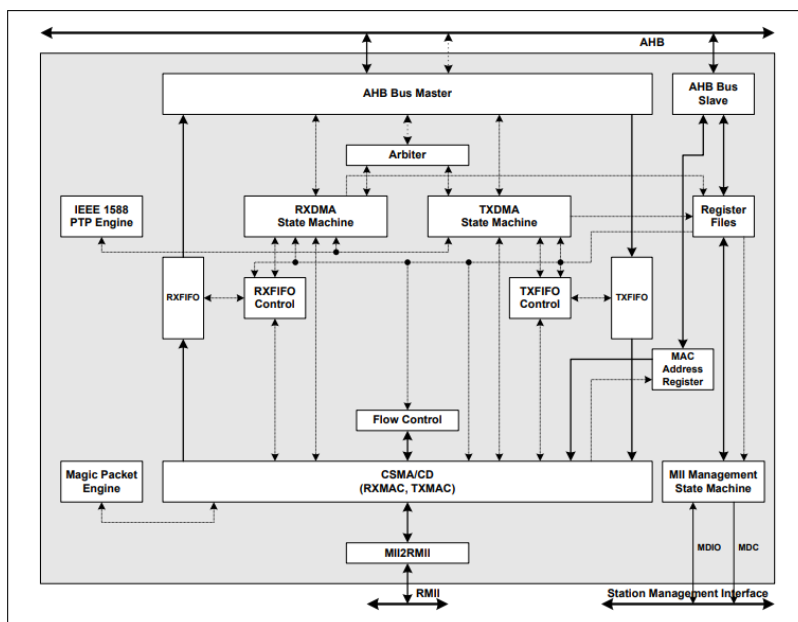
2.1. NUC980 serija mikroprocesora

NUC980 serija mikroprocesora zasnovana je na ARM926EJ-S jezgru. Postoji nekoliko mikroprocesora iz date serije koji se razlikuju po broju pinova (pakovanju) i veličini interne DDR2 memorije. Mikroprocesori iz ove serije posebno su opremljeni velikim brojem Ethernet, UART, CAN i sličnih periferija koji ih čine dobrim izborom za industrijsku i IoT primjenu [2]. Na slici 2 prikazana je blok šema strukture NUC980 mikroprocesora koja daje detaljniji prikaz resursa i mogućnosti koje posjeduju mikroprocesori iz date serije.



Slika 2.2 – Blok šema mikroprocesora iz NUC980 serije [3]

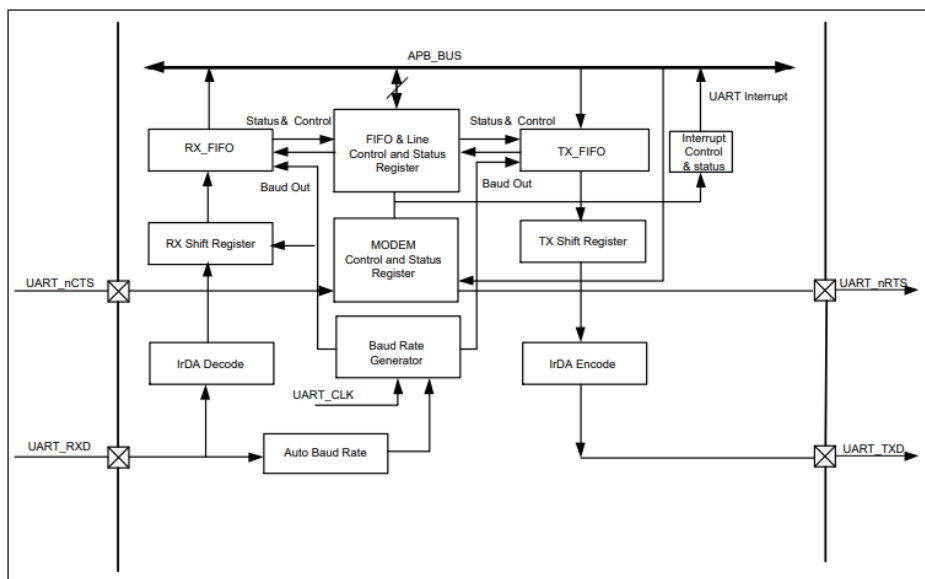
Za izradu ovog projektnog zadatka od najvećeg interesa je hardverska podrška za Ethernet (fizički sloj IEC 104 prokola zasnovan je na Ethernet-u) i podrška za UART, odnosno RS-485 (fizički sloj MODBUS RTU protokola zasniva se na nekom od serijskih protokola, u ovom slučaju RS-485). Što se tiče podrške za Ethernet, mikroprocesori iz NUC980 serije imaju dva Ethernet MAC kontrolera čime je hardverski podržana Ethernet komunikacija. Na slici 2.3 prikazana je struktura Ethernet kontrolera mikroprocesora iz NUC980 serije.



Slika 2.3 – Ethernet MAC kontroler NUC980 serije mikroprocesora [3]

Prvi Ethernet MAC kontroler zauzima prvih 10 pinova iz “E” banke pinova (E0 – E9), a drugi kontroler prvih 10 pinova iz “F” banke pinova (F0 – F9) [3]. Poznavanje ovog pinout-a je od značaja u sledećem koraku (priprema Linux operativnog sistema) kada se konfigurše struktura stabla uređaja (*device tree*). Ethernet portovi na ciljnoj ploči povezani su preko odgovarajućeg hardvera na ove pinove, čime je obezbeđen interfejs za Ethernet komunikaciju (u slučaju datih portova brzina komunikacije je 100 Mbps).

NUC980 serija mikroprocesora obezbeđuje čak deset kanala za UART komunikaciju pri čemu je hardverski podržan veliki broj konfiguracija i režima rada, između ostalog i željeni RS-485 režim. Blok šema UART kontrolera NUC980 serije mikroprocesora data je na slici 2.4.



Slika 2.4 – UART kontroler NUC980 serije mikroprocesora [3]

Za razliku od Ethernet kontrolera čiji su korišteni pinovi mikroprocesora jasno definisani, kod UART kontrolera RX i TX linije svakog kanala mogu da se konfiguriraju tako da se podese različiti korišteni pinovi, poznatije kao remapiranje pinova (*pin mapping*). Na primjer, kanal UART1 može za RX liniju da koristi pinove F9, A0 ili C6, dok kanal TX može da koristi F10, A1 ili C5. Dostupni pinovi za ostale kanale i linije jasno su navedeni u dokumentaciji proizvođača [3]. Što se tiče ciljne platforme, ono što je izazov jeste činjenica da je ona komercijalni proizvod te da dokumentacija, u kojoj je između ostalog i električna šema, nije javno dostupna. Da bi se pronašlo koji UART kanali se koriste kao i koji pinovi primjenjen je postupak reverznog inženjeringa, koji je podrazumijevao detekciju kratkog spoja između UART interfejsa na ploči i odgovarajućih pinova mikroprocesora pomoću multimetra. Nakon sprovedene procedure došlo se do informacije o korištenim UART kanalima kao i pinovima koji su u upotrebi. Ciljna platforma nudi šest RS-485 priključaka koji koriste šest UART kanala mikroprocesora. Korišteni kanali i konkretni pinovi mikroprocesora dati su u tabeli 2.1. Ono što se može napomenuti jeste da ciljna platforma koristi jedan UART kanal (UART0) u svrhe *debug* USB porta preko koga se serijskom vezom može pratiti proces učitavanja sistema te vršiti kontrola njegovog rada. Preko ovog priključka se takođe može vršiti

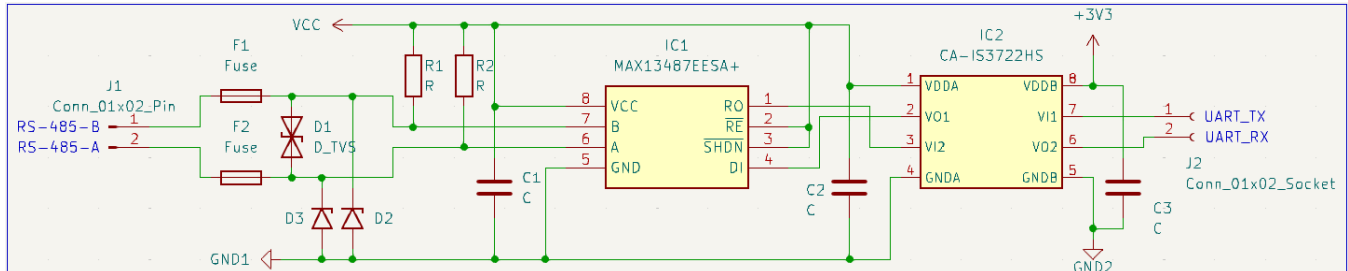
programiranje *flash* memorije ciljne platforme, međutim ona ne dolazi sa potrebnim prekidačima da bi se uvela u taj režim rada. Zbog toga se moralo pristupiti drugom načinu prenosa aplikacije na ciljnu platformu, koji će biti kasnije pojašnjen detaljnije.

Tabela 2.1 - Korišteni UART kanali i pinovi na ciljnoj platformi

Kanal	RX pin	TX pin
UART1	A0	F10
UART2	A9	A10
UART3	D3	D2
UART4	D13	D12
UART6	A4	A5
UART8	A11	A12

Kao što je već rečeno, poznavanje rasporeda pinova koji su iskorišteni je neophodno u fazi razvoja Linux operativnog sistema, odnosno za kreiranje strukture stabla uređaja. Ono što je još važno napomenuti prilikom razmatranja hardvera ciljne platforme jeste da što se tiče serijskog interfejsa MODBUS RTU protokola – RS-485, ciljna platforma sadrži 6 priključaka koji odgovaraju diferencijalnim A i B linijama RS-485. Signali sa ovih linija prolaze kroz odgovarajuću električnu mrežu da bi se doveli prema RX i TX linijama UART modula mikroprocesora. Električna mreža koja se nalazi između ima za ulogu da RS-485 diferencijalni signal (koji je većeg napona) obradi i konvertuje u +3.3V UART signal koji očekuje mikroprocesor. Prvi dio ove mreže nakon priključaka za RS-485 linije jeste sigurnosni i zaštitni dio koji se u slučaju ciljne platforme sastoji od redno vezanog osigurača i dva reda zaštitnih (TVS) dioda. Obje signalne linije RS-485 protokola (i A i B) prolaze kroz ovaj dio mreže, te idu na posebno integrisano kolo MAX13487E. Ovo kolo ima glavnu ulogu konverzije signala iz RS-485 oblika u signal koji se očekuje na UART linijama, i obrnuto. Dakle, kolo MAX13487E je RS-485 kolo primopredajnika zaduženo za konverziju oblika signala iz RS-485 u standardni UART, i obrnuto [4]. Treći i konačni korak obrade signala u datoj mreži jeste integrisano kolo CA-IS3722HS proizvođača *Chipanalog*. Ovo kolo je digitalno dvoulazno CMOS kolo koje služi za galvansku izolaciju ulaza i izlaza. Ovo kolo visokih performansi podržava brzinu prenosa podataka do 150 Mbps, specijalno namijenjeno u slučaju upotrebe za konverzije naponskih nivoa između serijskih protokola gdje su sa ulazne strane protokoli kao RS-485, CAN i slični [5]. U slučaju ciljne platforme, signali sa kola MAX13487E (sa pinova DI i RO) dovode se na pinove kola

CA-IS3722HS gdje se galvanski izoluju sa pinovima na drugoj strani. Ovim se obezbjeđuje da pinovi koji imaju visoke naponske nivoe sa RS-485 strane galvanski izoluju i spuste na nivoe koje odgovaraju UART modulu mikroprocesora. Naravno važi i obrnut proces, kod koga se UART signali poslati sa mikroprocesora konvertuju u RS-485 kompatibilan signal preko iste ove mreže. Na slici 2.5 prikazana je „gruba“ električna šema koja odgovara prethodno opisanoj mreži.



Slika 2.5 – Okvirna električna šema konverzije signala iz RS-485 u UART

Prethodnom diskusijom pokrivena je hardverska analiza ciljne platforme. Naredni korak je priprema operativnog sistema koji će se izvršavati na ploči, a upravo to je tema narednog poglavlja.

3. Priprema operativnog sistema za ciljnu platformu

Kao operativni sistem (OS) izabran je Linux OS kao najčešće korišten operativni sistem u domenu ugrađenih računarskih sistema. Razvoj aplikacija koje se izvršavaju na Linux operativnom sistemu sa sobom nose veliku prednost u odnosu na *bare-metal* pristup razvoja, a to je portabilnost. Naime, aplikacije razvijane za neki operativni sistem mogu da se pokreću na bilo kojoj ciljnoj platformi, nezavisno od hardvera, sve dok taj hardver podržava izvršavanje Linux operativnog sistema na njemu i ima dovoljno resursa koje zahtjeva data aplikacija. Pored samog operativnog sistema, za razvoj aplikacija potrebno je na razvojnoj platformi imati odgovarajući "alat" - *toolchain*. *Toolchain* predstavlja skup alata neophodnih za razvoj i kreiranje izvršnih fajlova koji se mogu pokrenuti na ciljnoj platformi. Osnovno sadrži alat kroskompajler (*cross-compiler*), ali i neke druge alate. S obzirom na sve potrebno da bi se i ciljna i razvojna platforma pripremile za razvoj aplikacije, najbolji izbor jeste upotreba nekog *build* sistema koji na relativno jednostavan način može da generiše sve što je neophodno.

3.1. Buildroot

U slučaju ovog projektnog zadatka korišten je *build* sistem *buildroot*, odnosno, tačnije rečeno, *fork* repozitorijuma ovog *build* sistema koji je napravio proizvođač mikroprocesora, kompanija Nuvoton [6]. *Buildroot* je kompletan *build* sistem alat koji može da generiše sve neophodne artefakte za razvoj aplikacija na ugrađenom sistemu - *toolchain*, Linux jezgro, *bootloader*, korjeni fajl sistem (*root file system*) i sve ostalo. Kao i većina sistema, koristi *Make* alat za izgradnju. Ovaj alat je relativno jednostavniji *build* sistem pogodan za projekte manjih i srednjih razmjera namjenjenih za ugrađene računarske sisteme bazirane na Linux jezgru. Pored generisanja osnovnih artefakata potrebnih za učitavanje na ciljnu platformu i razvoj aplikacija, *buildroot* alat nudi mogućnost konfiguracije, kako samog Linux jezgra, tako i kompletnog sistema dodavanjem raznih biblioteka i paketa neophodnih za razvoj aplikacije [7]. Ove konfiguracije zasnivaju se na veoma poznatom *Kconfig* alatu.

Početak razvoja počinje kloniranjem pomenutog *fork*-a repozitorijuma *buildroot*-a lokalno na razvojnoj platformi [6]. S obzirom da je *fork* napravio proizvođač, u njemu se nalaze početne konfiguracije koje se mogu primjeniti tako da se konfiguracija *buildroot*-a olakša u smislu da se jednostavnom primjenom konfiguracije i izgradnjom već dobijaju artefakti Linux jezgra koji se mogu pokrenuti na ciljnoj platformi, bez potrebe da se ručno modifikuju neke konfiguracione opcije koje se tiču glavnog hardvera ciljne platforme (procesori, arhitektura procesora, podržane *floating point* jedinice, ...). S obzirom da se koristi *make* alat, prvi korak jeste učitavanje neke od početnih konfiguracija komandom *make <naziv_početne_konfiguracije>_defconfig*. U slučaju ovog projektnog zadatka iskorištena je početna konfiguracija pod nazivom *nuvoton_nuc980_iot_defconfig*. Nakon učitavanja početne konfiguracije, komandom *make menuconfig*, može se pristupiti detaljnijoj konfiguraciji sistema. U slučaju ovog projektnog zadatka, u ovom dijelu konfiguracije odabrani su samo još neki dodatni paketi i biblioteke potrebni za izradu projektnog zadatka. Ove biblioteke i paketi navedeni su u nastavku:

- *libmodbus* (**BR2_PACKAGE_LIBMODBUS**) - jednostavna i široko poznata biblioteka koja implementira funkcionalnosti MODBUS protokola,
- *libjansson* (**BR2_PACKAGE_JANSSON**) - *lightweight* biblioteka čija je svrha jednostavno parsiranje i kreiranje *json* fajlova (kasnije će biti detaljnije pojašnjena uloga ovih fajlova u projektu),
- *openssh* (**BR2_PACKAGE_OPENSSSH**) - biblioteka potrebna za realizaciju *ssh* konekcije, koristi se u fazi razvoja aplikacije za mrežno povezivanje razvojne i ciljne platforme u cilju prenosa fajlova i udaljene kontrole ciljne platforme,
- *gpio* i *iio* alati (**BR2_PACKAGE_LINUX_TOOLS_GPIO** i **BR2_PACKAGE_LINUX_TOOLS_IIO**) - korisni alati koji nude pristup hardverskim pinovima razvojne platforme i njihovu kontrolu na visokom nivou iz korisničkog prostora jezgra.

Nakon ovih konfiguracija, pokretanjem komande *make* pokreće se izgradnja svih pomenutih artefakata. Po završetku procesa izgradnje na razvojnoj platformi su dostupni svi alati neophodni za razvoj aplikacija na ciljnoj platformi (*toolchain*), kao i slika jezgra, binarni fajl strukture stabla uređaja, *bootloader*, korjeni fajl sistem i sve ostalo (ovi fajlovi nalaze se na putanji *output/images* iz lokacije *buildroot* foldera). Još je ostalo da se razjasni kako se generisani fajlovi mogu programirati na ciljnu platformu, tako da se omogući učitavanje Linux jezgra i rad sa platformom. U opštem slučaju (npr. razvojne ploče kompanije Nuvoton), programiranje memorije ploče sa ovim fajlovima može se izvršiti upotrebom programatorskog alata kompanije Nuvoton - *NuWriter*. Da bi se omogućilo programiranje mikroprocesora, odnosno odgovarajuće *flash* memorije povezane sa mikroprocesorom, dva BOOT pina ploče (PG1 i PG0) moraju biti povezana u odgovarajući režim. Taj režim je režim USB BOOT, gdje oba pina moraju biti na niskom logičkom nivou. Ovo podešavanje je hardversko, odnosno razvojne ploče tipično sadrže odgovarajuće prekidače kojima se konfigurišu naponski nivoi ova dva pina. Nakon što se pristupi USB BOOT režimu, ploča se može programirati korištenjem slike *bootloader*-a, Linux jezgra i binarnog fajla stabla uređaja (*.dtb* fajl) preko pomenutog alata za programiranje [8]. Nakon programiranja, BOOT pinovi se vraćaju u drugi režim (*boot* preko odgovarajuće memorije, *flash* ili *spi flash*) te se učitavaju novoprogramirane binarne slike *bootloader*-a i Linux jezgra. Takođe, moguća je i opcija učitavanja preko SD/eMMC kartice. Ciljna platforma nema potrebne prekidače za podešavanje odgovarajućeg režima *boot*-a. Dakle, ponovo se nailazi na određene probleme koje samo po sebi nosi korištenje ciljne platforme razvijene u komercijalne svrhe. Da bi se prevazišao ovaj problem, primjenjena je sledeća ideja. Preko USB porta za *debug* prati se proces *boot*-ovanja ciljne platforme sa komercijalno razvijenim sistemom. Ovaj proces se sastoji iz dva glavna koraka. Prvo nastupa *bootloader* koji je u ovom slučaju vrlo poznat u oblasti ugrađenih sistema - *U-Boot*, nakon čega nastupa učitavanje Linux jezgra. Proces podizanja sistema može se prekinuti na pola, odnosno, može se prekinuti tokom izvršavanja *U-Boot*-a čime se pristupa *U-Boot* komandnoj konzoli. *U-Boot* je veoma moćan *bootloader* koji nudi razne mogućnosti tokom svog izvršavanja. Jedna od tih mogućnosti jeste konfiguracija mreže (podešavanje IP,

MAC adrese, adrese servera, gejtveja i ostalo) i korišćenje nekih osnovnih protokola, kao što je *Trivial File Transfer Protocol (TFTP)*. U *U-Boot* na ciljnoj platformi potrebno je podesiti varijable okruženja za IP adresu (*ipaddr*), adresu servera - razvojna platforma (*serverip*) i MAC adresu *emac* kontrolera (*ethaddr*). Na razvojnoj platformi potrebno je osposobiti TFTP server, te u folder na kome tftp server očekuje fajlove za prenos, kopirati sliku Linux jezgra (*uImage*) i binarnu sliku stabla uređaja (*.dtb* fajl). Nakon ovoga, moguće je iz *U-Boot*-a na ciljnoj platformi učitati date fajlove putem TFTP protokola komandom *tftp* u radnu memoriju, te izvršiti proces podizanja Linux jezgra komandom *bootm* [9]. Ovaj način podizanja operativnog sistema na ugrađenim računarskim sistemima poznat je i kao podizanje sistema preko mreže i iskorišten je kao najjednostavnije rješenje da se premosti prethodno opisani problem programiranja ciljne platforme koji je dovoljno dobar u fazi razvoja i testiranja sistema.

3.2. Konfiguracija Linux jezgra, Linux rukovaoci uređaja, struktura stabla uređaja

Postupkom opisanim u prethodnoj podglavi dobijaju se svi potrebni fajlovi za podizanje Linux jezgra na ciljnoj platformi. Što se tiče samog Linux jezgra, *buildroot* alat koristi takođe specijalni *fork* Linux jezgra koji je takođe obezbjedila kompanija Nuvoton. Dato Linux jezgro se postupkom mrežnog podizanja sistema može bez problema izvršavati na ciljnoj platformi, čime se obezbjedio glavni uslov za izradu aplikacije projektnog zadatka. Međutim, ono o čemu se mora voditi računa jeste da tako generisano Linux jezgro možda ne podržava sav hardver ili periferije neophodne za realizaciju zahtjevanih projektnih zadataka. U drugoj glavi opisan je potreban hardver koji podrazumijeva dva Ethernet porta i šest UART kanala (pored, naravno, nekih dodatnih hardverskih modula kao što je npr. DMA kontroler). Da bi se omogućili, konfigurisali i uspješno koristili neophodni hardverski moduli neophodno je obratiti pažnju na dvije komponente Linux jezgra, a to su rukovaoci uređaja - drajveri (*drivers*) i struktura stabla uređaja (*device tree*).

3.2.1. Rukovaoci uređaja (*Linux Device Drivers*)

Rukovaoci uređaja su jedan od centralnih pojmova Linux operativnog sistema. Njihova uloga jeste da korisnicima (programerima i njihovim aplikacijama) obezbjedi jednostavan i dobro poznat interfejs kojim korisnička aplikacija može da interaguje sa određenim hardverom sistema na kome se izvršava Linux jezgro. Sva kompleksnost i pojedinosti datog hardvera skrivene su od korisničke aplikacije, ona samo koristi poznati interfejs (*API*) koji obezbjeđuje drajver. Drajveri su zaduženi da zahtjeve na visokom nivou koji dolaze od korisničkih aplikacija implementiraju te direktno interaguju sa hardverom za koji su zaduženi u svrhu izvršenja tih zahtjeva [10].

Omogućavanje odgovarajućih drajvera prilikom konfiguracije i izgradnje Linux jezgra omogućeno je takođe upotrebom *Kconfig* alata. Što se tiče ovog projektnog zadatka, potrebno je da u konfiguraciji omogućimo sve neophodne drajvere za Ethernet, kao i za UART komunikaciju. Koristeći *buildroot* alat, ovim konfiguracijama možemo pristupiti, i mijenjati ih, putem komande *make linux-menuconfig*. Potrebni drajveri koje treba omogućiti (ugraditi u sliku jezgra ili kao module) dati su u nastavku [11]:

- ICPlus PHYs (**CONFIG_ICPLUS_PHY**) - drajver za fizički sloj, neophodan za Ethernet komunikaciju,
- Nuvoton NUC980 Ethernet MAC 0 (**CONFIG_NUC980_ETH0**) - drajver prvog Ethernet porta,
- Nuvoton NUC980 Ethernet MAC 1 (**CONFIG_NUC980_ETH1**) - drajver drugog Ethernet porta i
- NUC980 UARTx Support (**CONFIG_NUC980_UARTx**) - drajveri za UART serijsku komunikaciju, pri čemu je 'x' broj UART kanala koji se koristi, a koji se nalaze u tabeli 2.1.

Rukovaoci uređaja su neophodni da bi korisničke aplikacije pristupile hardveru od interesa, međutim, oni ne mogu sami u potpunosti da obave posao. Naime, drajveri moraju nekako da dobiju informaciju o tome gdje su odgovarajuće periferije povezane sa mikroprocesorskom jedinicom i na koji način, kao i da dobiju informaciju o nekim osnovnim parametrima i konfiguracijama hardverskih elemenata sistema. Ove informacije drajveri dobijaju od drugog veoma bitnog dijela Linux sistema - strukture stabla uređaja (*device tree*).

3.2.2. Stablo uređaja (*Device Tree*)

Stablo uređaja predstavlja strukturu podataka i jezik koji služi za opis hardvera sistema. Kao što ime kaže, struktura izvornih fajlova (*.dts* fajlovi) sastoji se od čvorova (koji predstavljaju hardverske entitete sistema) koji se dodaju hijerarhijski u obliku stabla, onako kako bi se to očekivalo u hardverskoj hijerarhiji. Način kreiranja i pisanja čvorova za neki hardver ili periferiju koja je povezana na sistem, definisan je opštim pravilima koja se nazivaju *Device Tree Bindings* (*dt-bindings*). Jedan od glavnih ciljeva upotrebe ove strukture jeste da se drajverima uređaja omogući jedan uniforman i jednostavan način za konfiguraciju određenih parametara hardverskog entiteta kojim upravlja, te informacije o hardverskim vezama mikroprocesora i datog entiteta. Sve ove informacije drajver uređaja dobija preko njemu odgovarajućeg čvora u izvornom fajlu stabla uređaja (definisano *compatible* poljem). Ovim se postiže elegantan način konfiguracije drajvera i hardvera tokom podizanja i rada samog Linux sistema, bez potrebe da se ovakve stvari "*hard* koduju" u izvorni kod drajvera, te mijenjaju svaki put kada treba neka promjena što bi takođe zahtjevalo ponovnu gradnju Linux jezgra [12].

Što se tiče ovog projektnog zadatka, neophodno je podesiti čvorove stabla uređaja koji se tiču Ethernet kontrolera i odgovarajućih UART modula. Svi ovi čvorovi kao i ostale bitne konfiguracije, već su kreirani u krovnom *nuc980.dtsi* fajlu za mikroprocesor NUC980 koji je obezbijedila kompanija Nuvoton. Na korisniku ostaje da kreira zaseban izvorni fajl stabla uređaja, u kom će da iskoristi kreirane čvorove, omogući ih kroz polje *status* i, ukoliko je to neophodno, podesi odgovarajuće pinove na koje su povezane periferije. U nastavku prvo će biti prikazan način na koji se preko strukture stabla uređaja daju informacije o pinovima. Ovo je moguće preko čvora koji se obično naziva *pinctrl*. Unutar ovog čvora se definišu podčvorovi koji odgovaraju periferijama, te unutar tih čvorova se kreiraju konačni čvorovi koji predstavljaju konfiguraciju pinova. U drugoj glavi razmotreni su pinovi na koje su povezani Ethernet

portovi - pinovi E banke od E0 do E9 za prvi, odnosno F banka od F0 do F9 za drugi port. Ovakvu konfiguraciju možemo uočiti ako pogledamo *emac0* i *emac1* podčvorove čvora *pinctrl* što je prikazano na slici 3.1.

```
emac0 {
    pinctrl_emac0: emac0 {
        nuvoton,pins =
            <4 0 1 0
            4 1 1 0
            4 2 1 0
            4 3 1 0
            4 4 1 0
            4 5 1 0
            4 6 1 0
            4 7 1 0
            4 8 1 0
            4 9 1 0>;
    };
};

emac1 {
    pinctrl_emac1: emac1 {
        nuvoton,pins =
            <5 0 1 0
            5 1 1 0
            5 2 1 0
            5 3 1 0
            5 4 1 0
            5 5 1 0
            5 6 1 0
            5 7 1 0
            5 8 1 0
            5 9 1 0>;
    };
};
```

Slika 3.1 – *pinctrl* podčvorovi za Ethernet kontrolere

Sa slike se može uočiti da se koristi konstrukcija *nuvoton,pins* koja očekuje nizove konfiguracija pina od po 4 vrijednosti koje predstavljaju odgovarajuće osobine pina. Prva vrijednost je broj banke pina pri čemu se kreće od banke A (vrijednost 0) do banke G (vrijednost 6). Druga vrijednost je indeks pina iz date banke, pri čemu svaka banka ima ukupno 16 pinova (indeksi od 0 - 15). Treća opcija je vrlo važna i označava funkciju pina koja se može odabrati konceptom remapiranja. Konkretnu vrijednost i funkcije pina vezane za njih dati su u dokumentaciji proizvođača [3]. Četvrta vrijednost su neke dodatne osobine konkretnog pina, i ova vrijednost se rijetko unosi [11]. Na slici 3.2 prikazan je primjer podčvora *uart1* čvora *pinctrl*.

```
uart1 {
    pinctrl_uart1_PF: uart1-PF {
        nuvoton,pins =
            <5 7 2 0
            5 8 2 0
            5 9 2 0
            5 0xa 2 0>;
    };

    pinctrl_uart1_PA: uart1-PA {
        nuvoton,pins =
            <0 0 4 0
            0 1 4 0>;
    };

    pinctrl_uart1_PC: uart1-PC {
        nuvoton,pins =
            <2 5 7 0
            2 6 7 0
            2 7 7 0
            2 8 7 0>;
    };
};
```

Slika 3.2 – Podčvor *uart1* čvora *pinctrl*

Sa slike može se primjetiti da postoje neke predefinisane konfiguracije pinova, s obzirom na spomenute mogućnosti remapiranja koje su razmotrene u drugoj glavi. Ono što se može primjetiti jeste da nema definisane konfiguracije za onu koja je potrebna u slučaju hardverskih veza ciljne platforme (pogledati tabelu 2.1). U ovom slučaju, taj čvor sa željenom konfiguracijom pinova će biti neophodno dodati u izvorni fajl stabla uređaja koji se korisnički definiše, što će biti prikazano kasnije. Na sljedećim slikama 3.3 i 3.4 su prikazani primjeri konkretnih čvorova Ethernet i UART kanala koji će biti proslijeđeni drajverima radi konfiguracije, i koji se mogu naći u pomenutom *.dtsi* fajlu.

```
emac0@b0012000 {
    compatible = "nuvoton,nuc980-emac0";
    reg = <0xb0012000 0x1000>;
    interrupts = <21 4 1>, <19 4 1>;
    pinctrl-names = "default";
    pinctrl-0 = <&pinctrl_emac0>;
    status = "okay";
};

emac1@b0022000 {
    compatible = "nuvoton,nuc980-emac1";
    reg = <0xb0022000 0x1000>;
    interrupts = <22 4 1>, <20 4 1>;
    pinctrl-names = "default";
    pinctrl-0 = <&pinctrl_emac1>;
    status = "okay";
};
```

Slika 3.3 – Čvorovi Ethernet kontrolera


```
uart1: serial@b0071000 {
    compatible = "nuvoton,nuc980-uart";
    reg = <0xb0071000 0x1000>;
    interrupts = <37 4 1>;
    port-number = <1>;
    map-addr = <0xf0071000>;
    pinctrl-names = "default";
    pinctrl-0 = <&pinctrl_uart1_PA>;
    pdma-enable = <0>;
    status = "disabled";
};

uart2: serial@b0072000 {
    compatible = "nuvoton,nuc980-uart";
    reg = <0xb0072000 0x1000>;
    interrupts = <38 4 1>;
    port-number = <2>;
    map-addr = <0xf0072000>;
    pinctrl-names = "default";
    pinctrl-0 = <&pinctrl_uart2_PA>;
    pdma-enable = <0>;
    status = "disabled";
};
```

Slika 3.4 – Primjeri čvorova UART kanala

Ono što ostaje da se uradi u glavnom *.dts* fajlu jeste da se odgovarajući čvorovi iz ovog fajla referensiraju, omoguće i po potrebi dodatno konfigurišu tako da odgovaraju stvarnom stanju hardvera na ciljnoj platformi. Konačno, na slici 3.5 je dat isječak iz glavnog *.dts* fajla u kome su konfigurisani svi potrebni UART kanali koji se koriste na ciljnoj platformi. Sa slike se može primjetiti da je dodana odgovarajuća konfiguracija pinova koje koristi kanal UART1, pošto ista nije bila dostupna u *.dtsi* fajlu. Nakon toga su omogućeni potrebni UART kanali, pri čemu je za svaki kanal odabrana ona konfiguracija pinova koja odgovara stvarnim vezama na ciljnoj platformi. Kompletan *.dts* fajl sadrži dodatne čvorove koji su referensirani iz glavnog fajla i koji su onemogućeni (npr. I2C i SPI čvorovi) jer nisu potrebni za konkretan projekat.

Prethodno opisanom procedurom dobija se pripremljena konfiguracija Linux jezgra na kojoj je omogućen i podešen za rad sav zahtjevani hardver koji se nalazi na ciljnoj platformi. Do pokretanja *make* komande i dobijanja novih slika jezgra ostaje samo još jedan korak, koji nije obavezan, ali za konkretan sistem je preporučljiv.

```
pinctrl: pinctrl@b0000000 {
    uart1 {
        pinctrl_uart1_PA_PF: uart1-PA-PF {
            nuvoton,pins =
                <0 0 4 0
                | 5 0xa 2 0>;
        };
    };
};

uart1: serial@b0071000 {
    pinctrl-names = "default";
    pinctrl-0 = <&pinctrl_uart1_PA_PF>;
    pdma-enable = <1>;
    status = "okay";
};

uart2: serial@b0072000 {
    pinctrl-names = "default";
    pinctrl-0 = <&pinctrl_uart2_PA>;
    pdma-enable = <1>;
    status = "okay";
};

uart3: serial@b0073000 {
    pinctrl-names = "default";
    pinctrl-0 = <&pinctrl_uart3_PD>;
    pdma-enable = <1>;
    status = "okay";
};

uart4: serial@b0074000 {
    pinctrl-names = "default";
    pinctrl-0 = <&pinctrl_uart4_PD>;
    pdma-enable = <1>;
    status = "okay";
};

uart6: serial@b0076000 {
    pinctrl-names = "default";
    pinctrl-0 = <&pinctrl_uart6_PA>;
    pdma-enable = <1>;
    status = "okay";
};

uart8: serial@b0078000 {
    pinctrl-names = "default";
    pinctrl-0 = <&pinctrl_uart8_PA_PG>;
    pdma-enable = <1>;
    status = "okay";
};
```

Slika 3.5 – Isječak glavnog .dts fajla

3.2.3. Konfiguracija Linux jezgra za rad u realnom vremenu

Kao što je već rečeno, gejtvej je mrežni uređaj koji spada u grupu ugrađenih računarskih sistema. Ovakvi uređaji često imaju zahtjeve za rad u realnom vremenu i često se smatraju kao ugrađeni sistemi koji rade u realnom vremenu. Zbog toga se kao poslednji korak u pripremi Linux operativnog sistema izvodi njegova konfiguracija za rad u realnom vremenu, poznata kao “pečiranje” Linux jezgra za rad u realnom vremenu (*Real-time patch*). Ova procedura podrazumijeva preuzimanje dostupnog fajla zakrpe (*patch file*) dostupnog onlajn, te njegovu primjenu na fajlove Linux jezgra primjenom komande *patch -p1* unutar korjenog foldera Linux jezgra [13]. Treba voditi računa da fajl zakrpe mora odgovarati verziji Linux jezgra koji se izgrađuje (u ovom slučaju verzija je 5.10.140). Fajl zakrpe modifikuje razne konfiguracione fajlove unutar izvornog koda jezgra čime omogućava njegovu konfiguraciju kao sistem za

rad u realnom vremenu. Ove konfiguracije mijenjaju mogućnosti nekih standardnih biblioteka (na primjer biblioteke *pthread*, od koje uveliko zavisi biblioteka *lib60870*), ali ono što je najvažnije, podešavaju raspoređivač Linux jezgra tako da podrži *hard* zahtjeve za rad u realnom vremenu. Nakon primjene zakrpe, potrebno je samo omogućiti konfiguracionu opciju **CONFIG_PREEMPT_RT** čime je Linux jezgro potpuno prilagođeno za rad u realnom vremenu.

Nakon ove procedure, preostaje da se ponovo izgradi Linux jezgro upotrebom komande *make linux-rebuild*, te konačno izgradnja svih fajlova upotrebom *buildroot* alata pokretanjem komande *make*. Novonastala slika kernela i *.dtb* fajl mogu se učitati preko mreže i iskoristiti za podizanje jezgra na ciljnoj platformi. Ovim se završava sva procedura generisanja neophodnih resursa za razvoj aplikacije, kako na razvojnoj, tako i na izvršnoj platformi.

4. Razvoj aplikacije projektnog zadatka

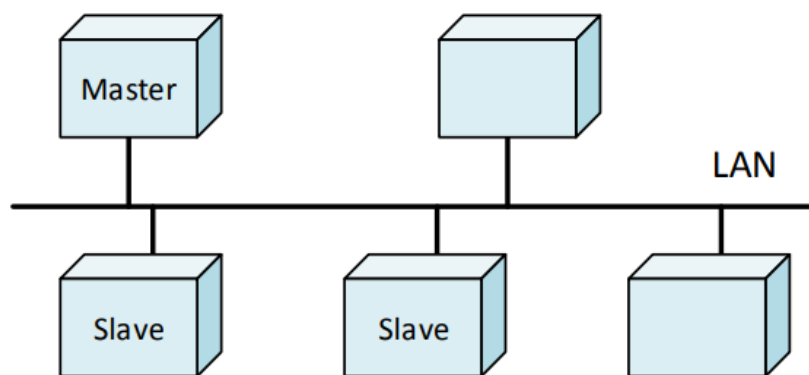
Nakon uspješne pripreme razvojne i ciljne platforme može se preći na konačni korak, a to je izrada same aplikacije koja će obavljati zahtjeve tražene projektnim zadatkom. Dakle, osnovni cilj aplikacije jeste implementacija *gateway* uređaja između dva industrijska protokola: IEC 104 i MODBUS RTU. Prije nego što se pređe na detalje implementacije aplikacije, prikazaće se kratak pregled datih protokola, i biblioteka koje su korištene za implementaciju datih protokola u aplikaciji.

4.1. Protokol IEC 60870-5-104 (IEC 104)

IEC (*International Electrotechnical Commission*) 104 je industrijski protokol koji se koristi primarno u elektroenergetskim sistemima za kontrolu i nadzor električnih mreža u elektranama. Njegova uloga jeste prenos komandi i razmjena podataka od interesa u elektroenergetskim sistemima. Sam protokol zasnovan je na istom aplikativnom modelu kao i IEC 101 (protokol iz iste serije 60870-5), ali se od njega razlikuje po nižim slojevima komunikacije. IEC 101 koristi serijsku vezu na fizičkom sloju i specijalan sloj veze podataka, dok IEC 104 dodaje na aplikativni sloj zaglavlje koje je slično podacima sa IEC 101 sloja veze podataka, a za sve niže slojeve koristi standardni TCP/IP protokol. Ovim se praktično postiže realizacija IEC 101 protokola preko Ethernet veze što sa sobom nosi razne prednosti [14]. Kod IEC 104 (i 101) protokola komunikacija je definisana između dvije strane:

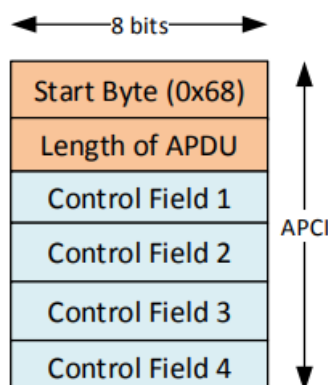
- Kontrolišuća stanica (*controlling station*), master stanica ili klijent – Uređaj koji kontroliše rad ostalih stanica u mreži slanjem komandi i prihvatanjem/zahtjevanjem podataka od datih stanica. Može biti npr. SCADA računar u elektroenergetskim sistemima.
- Kontrolisana stanica (*controlled station*), slave stanica ili server – jedan ili više senzorskih i/ili aktuatorskih uređaja (ili grupa istih) koji izvršavaju rad ili mjere određene veličine u sistemu. Kontrolišuća stanica od ovih uređaja uzima vrijednosti i šalje im odgovarajuće komande koje treba da se izvrše. Može biti npr. skup releja koji služe za otvaranje/zatvaranje ventila u elektroenergetskom sistemu.

U smislu ovog projektnog zadatka, gejtvej koji se implementira je IEC 104 **slave** uređaj. Odnosno on treba da komande od strane nekog master uređaja (npr. SCADA) prevede u MODBUS RTU komande i izvrši ih (naravno isto važi i za prenos podataka). IEC 101 definiše dva načina prenosa podataka: balansirani prenos i nebalansirani prenos. Kod nebalansiranog prenosa komunikacija uvijek kreće od master uređaja, dok slave uređaji odgovaraju. Kod balansiranog prenosa, slave uređaji mogu bez zahtjeva master uređaja da šalju poruke. Kod IEC 104 protokola podržan je samo balansirani režim. Što se tiče mrežnih topologija, tipično se koriste PTP (*Point To Point*) veze, MPTP (*Multiple Point To Point*) veze ili veze u topologiji magistrale [14]. Na slici 4.1 prikazan je primjer jedne topologije magistrale u IEC 104 protokolu. U slučaju ovog projektnog zadatka sa strane gejtveja, veza sa master stanicom je *Point To Point* Ethernet link.



Slika 4.1 – Topologija magistrale u IEC 104 protokolu [14]

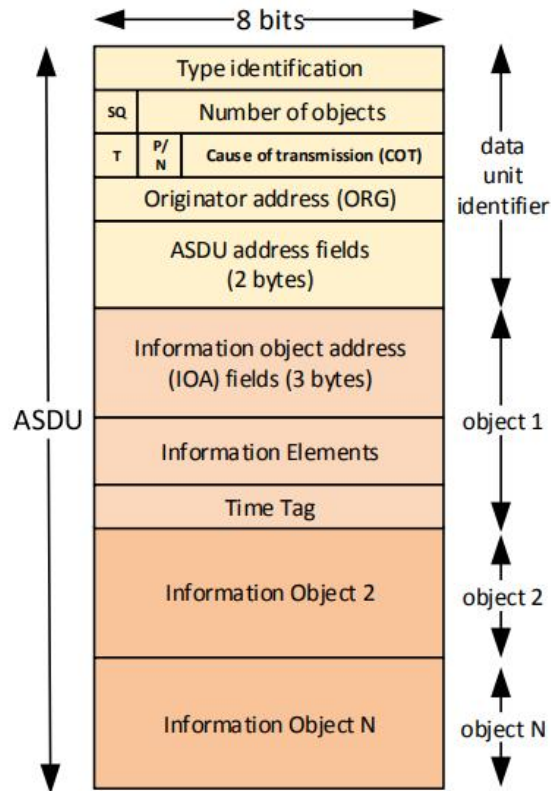
IEC 104 protokol na aplikativnom sloju definiše zaglavlje koje se naziva **APCI** (*Application Protocol Control Information*) nakon koga slijedi frejm podataka koji se naziva **ASDU** (*Application Service Data Unit*). Zajedno oni čine kompletan frejm aplikativnog sloja IEC 104 protokola koji se naziva **APDU** (*Application Protocol Data Unit*). Na slici 4.2 prikazano je APCI zaglavlje IEC 104 protokola.



Slika 4.2 – APCI zaglavlje IEC 104 protokola [14]

APCI se sastoji od šest polja od čega svako polje zauzima po jedan bajt. Prvo polje je konstantno i označava početak frejma aplikativnog sloja IEC 104 protokola. Sledeće polje je polje koje označava veličinu APDU i korisno je za izračunavanje veličine bloka podataka kasnije u frejmu. Naredna četiri polja su kontrolna polja koja imaju različito značenje u odnosu na format poruke. Postoje tri formata: S-format, U-format i I-format. I-format koristi sva četiri polja u svrhu sekvenciranja poruka (da se spriječi propuštanje poruka i *overflow* bafera podataka na bilo kojoj od strana u komunikaciji) i nakon ovog zaglavlja uvijek slijedi frejm podataka. S-format se koristi od strane slave uređaja samo u svrhu sekvenciranja u kome se potvrđuju primanja poruka. Nakon ovog zaglavlja ne idu podaci, već se koristi samo kada slave uređaj dugo nije potvrdio prijem poruke, pa master uređaj prestaje sa slanjem da se ne bi desila prekoračenja bafera. Poslednji, U-format, se koristi u svrhu slanja kontrolnih frejmova i nema ulogu

sekvenciranja. Zaglavljem ovog formata se najčešće označavaju početak i kraj razmjene između master i slave uređaja kao i označavanje testnih frejmova. Takođe ne sadrži frejm podataka nakon zaglavlja [14].



Slika 4.3 – ASDU format poruke IEC 104 protokola [14]

Na slici 4.3 prikazan je format ASDU poruke aplikativnog sloja IEC 104 protokola. Ovaj frejm je ključan za IEC 104 protokol jer u sebi nosi sve od interesa (komande, podatke, ...). Prvih 5 bajtova služe za identifikaciju jedinice podatka nakon čega kreću stvarni podaci od interesa, koji su u IEC 104 protokolu definisani kao **Informacioni objekti (Information Object)**. Prvo ćemo razmotriti dio frejma koji se odnosi na identifikaciju jedinice podatka [14]:

- Identifikator tipa (*Type identification*) – Koristi se za identifikaciju tipa podataka koji se šalju u konkretnom frejmu, odnosno koji su sadržani u informacionim objektima. Sam protokol definiše 128 mogućih identifikatora tipova pri čemu je moguće dodati i korisnički definisane. Ono što je važno za napomenuti jeste da u jednom ASDU može da se nađe više informacionih objekata, ali svaki od ovih objekata mora da sadrži ISTI tip podatka.
- SQ (*Structure Qualifier*) – Označava način na koji se informacioni objekti adresiraju unutar ASDU. Naime, ASDU može da ima više informacionih objekata koji u sebi nose podatke (istog tipa), pri čemu svaki informacioni objekat ima svoju adresu. Moguće je takođe da u ASDU bude samo jedan informacioni objekat, ali koji nosi sekvencu podataka (informacionih elemenata), pri

čemu se ovi podaci poredani tako da budu sekvencijalni, odnosno pristupa im se uzastopno jednostavnim *offset*-ovanjem onoliko bajta koliko zauzima odgovarajući tip podatka. Za slučaj sekvenciranog prenosa, ovaj fleg bit se postavlja na jedinicu.

- Broj informacionih objekata (*Number of Objects*) – Brojna vrijednost koja označava broj informacionih objekata u ASDU. Jedan ASDU može maksimalno da sadrži 127 informacionih objekata.
- T (*Test flag bit*) – ovaj fleg bit može da se postavi na jedinicu kada je ASDU koji se šalje testni i ne treba stvarno da promjeni stanje kontrolisane stanice kome se poruka šalje.
- P/N (*Positive/Negative flag bit*) – Ovaj fleg bit može da se iskoristi sa strane kontrolisane stanice da bi se označila uspješnost izvršavanja neke komande. Ako je ovaj fleg bit postavljen na jedinicu, odgovor je negativan što za kontrolišuću stanicu označava da prethodna komanda nije uspješno izvršena.
- Razlog prenosa poruke (*Cause Of Transmission - COT*) – Ovo polje je jedno od najvažnijih i karakterističnih polja IEC 104 protokola. Kao što samo ime kaže, označava razlog zbog koga se određena poruka šalje. Koristi se za kontrolu smjera poruka, dijelom označava željenu akciju te omogućava mehanizam prenosa razloga greške u izvršavanjima nekih komandi, ako se ona desi. Protokol definiše 48 standardnih vrijednosti za COT, pri čemu je moguće definisati dodatne.
- Adresa izvorišta (*Originator Address*) – Koristi se sa strane kontrolišuće stanice koja upisuje svoju adresu na ovo polje. Kontrolisana stanica uglavnom ne mijenja ovo polje što znači da ovo polje uglavnom označava adresu master uređaja u mreži. Ukoliko postoji samo jedna kontrolišuća stanica, ovo polje nije obavezno da se podešava te uvijek može biti nula.
- Adresa ASDU (*ASDU Address Field*) – Ovo polje označava adresu kontrolisane stanice kojoj se šalje data poruka. U fizičkom smislu može da odgovara stvarnoj jednoj stanici, ali moguće je i da se jedna stanica „podijeli“ na više logičkih cjelina od koje svaka ima svoju adresu. Master stanica postavlja ovo polje kada šalje poruku odgovarajućoj slave stanici, dok u slučaju slanja poruke sa slave stanice, ona upisuje svoju adresu na ovo polje. Adrese mogu da budu veličine 1 ili 2 bajta što je konfigurabilno na nivou čitave mreže (sistema).

Nakon što se podese polja iz dijela identifikacije jedinice podatka, slijede polja koja predstavljaju konkretne podatke. Prvo od tih polja je adresa informacionog objekta (*Information Object Address*). Ovo polje služi da adresira odgovarajući objekat kontrolišuće stanice koji sadrži podatak ili podatke od interesa. Ovi objekti se logički dijele na strani kontrolisane stanice prema svojim fizičkim osobinama i kontrolisana stanica ih prezentuje kontrolišućoj preko odgovarajućeg odgovora na specijani zahtjev (*Interrogation request*). Nakon ove adrese, slijedi konkretan podatak koji se u IEC 104 protokolu naziva informacioni element (*Information element*). Ukoliko se poruka šalje od strane kontrolišuće stanice, tada u slučaju komande podatak sadrži informacije o komandi koja treba da se izvrši (npr. ako je komanda postavljane vrijednosti u nekoj stanici, podatak može biti vrijednost koja se treba postaviti). U slučaju da se poruka šalje sa strane kontrolisane stanice, onda je podatak obično vrijednost koja se šalje (koja je zahtjevana,

koja se šalje periodično ili spontano). Postoji nekoliko standardnih podataka koji su definisani protokolom. Ovi podaci mogu u nekim slučajevima da dođu i sa vremenskim žigom (*Time Tag*).

Ovim izlaganjem dat je kratak pregled IEC 104 protokola koji je neophodan da bi se razumjela terminologija i ispravno korištenje biblioteke za implementaciju IEC 104 slave uređaja na gejtveju.

4.1.1. IEC 104 biblioteka – *lib60870*

Biblioteka *lib60870* razvijena je od strane organizacije *mz-automation* i u njoj se nalazi implementacija aplikativnog sloja IEC 104 i IEC 101 protokola (takođe, i sloja veze podataka IEC 101 protokola) u C programskom jeziku. Implementirane su funkcionalnost za sve strane komunikacije (i master i slave), kao i podrška za nebalansirani prenos za IEC 101 i dodatna podrška za sigurnost implementacijom TLS (*Transport Layer Security*). Biblioteka je implementirana principom slojevitog razvoja softvera tako da su prema korisniku biblioteke maksimalno apstrahovane i odvojene funkcionalnosti niskog nivoa (postoje gotove API funkcije za kreiranje niti, alociranje memorije, pristup hardveru, ...). Kao što je već rečeno, u svrhu ovog zadatka potrebno je implementirati IEC 104 slave funkcionalnosti na gejtveju. Biblioteka obezbeđuje odgovarajuće API funkcije kojima se može kreirati instanca IEC 104 slave uređaja (*CS104_Slave_create*) i podesiti mrežna konfiguracija i broj dozvoljenih konekcija (*CS104_Slave_setLocalAddress*, *CS104_Slave_setServerMode*). Takođe je obezbeđeno nekoliko struktura podataka koje se vežu uz instacu slave uređaja i kojima se mogu promijeniti neki parametri aplikativnog sloja (*CS104_APCIPParameters* i *CS101_AppLayerParameters*). Ključni korak pri implementaciji slave uređaja pomoću ove biblioteke jeste implementacija odgovarajućih *callback handler* funkcija koje biblioteka obezbeđuje. Biblioteka nudi korisniku (programeru) koji implementira slave uređaj način da odgovori na zahtjeve IEC 104 master uređaja putem implementacije odgovarajućih *handler* funkcija. Ove funkcije imaju odgovarajući prototip koji moraju poštovati, te se date funkcije moraju upotrebom odgovarajućih API funkcija biblioteke registrovati unutar slave instance (*CS101_Slave_set*Handler*) da bi se automatski aktivirali kada se preko mreže prihvati neki zahtjev od master strane. Dostupne *handler* funkcije koje se mogu implementirati date su na slici 4.4.

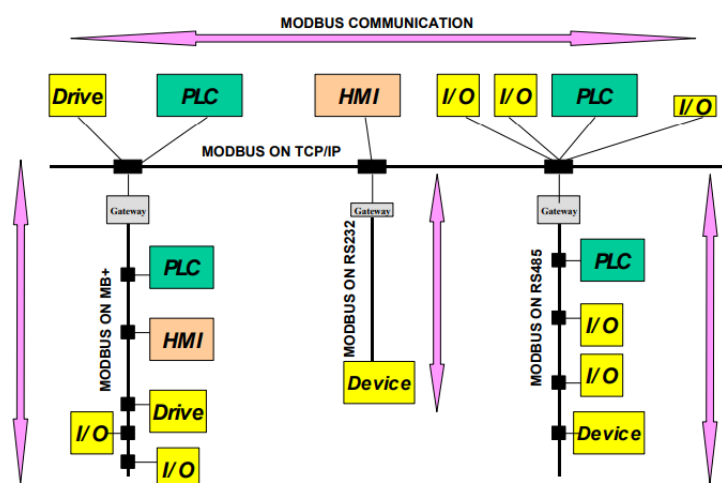
callback type	event	CS 101	CS 104
CS101_InterrogationHandler	interrogation requests	+	+
CS101_CounterInterrogationHandler	counter interrogation requests	+	+
CS101_ReadHandler	read requests for single information objects	+	+
CS101_ClockSynchronizationHandler	clock synchronization message received	+	+
CS101_ResetProcessHandler	reset process request received	+	+
CS101_DelayAcquisitionHandler	delay acquisition request received	+	-
CS101_ASDUHandler	ASDU received but not handled by one of the other callback handlers	+	+
CS101_ResetCUHandler	a link layer message of type reset CU (communication unit) has been received	+	-
CS104_ConnectionRequestHandler	a new TCP/IP client tries to connect	-	+

Slika 4.4 – *Callback handler funkcije biblioteke lib60870* [15]

Nakon kreiranja instance slave uređaja, podešavanja svih parametara i registracije *handler* funkcija, slave instanca mora da se startuje pozivom funkcije *CS104_Slave_start*. Odgovor na zahtjeve master uređaja (komande, zahtjevi za čitanjem podataka i ostali) slave realizuje putem *handler* funkcija, dok se izvršavanje glavnog programa može svesti na realizaciju slanja periodičnih poruka koje su definisane kao razlog prenosa (COT) unutar IEC 104 protokola. Što se tiče kreiranja konkretnih poruka, biblioteka nudi API funkcije kojima se mogu kreirati instance poruke - ASDU instance (*CS101_ASDU_create*), kreirati instance informacionih objekata (*InformationObject_create*) i konačno dodati objekte u sam ASDU (*CS101_ASDU_addInformationObject*). Biblioteka nudi razne *get* i *set* funkcionalnosti nad instancama ovih objekata (strukturama podataka) kojima mogu da se podeše pojedinačna polja ASDU, dodaju informacioni elementi u objekat i mijenjaju dodatna polja informacionih objekata. Informacioni objekti takođe imaju definisane konkretne strukture za postojeće tipove elemenata (*SinglePoint*, *DoublePoint*, *ScaledValue*, ...) i dodavanjem jednog ovakvog objekta u ASDU instancu, automatski se podešava polje identifikacije tipa. Svi detalji oko API funkcija biblioteke, implementacije *handler* funkcija, dostupnih tipova, struktura podataka i konstanti mogu se pronaći na *GitHub* stranici biblioteke [15].

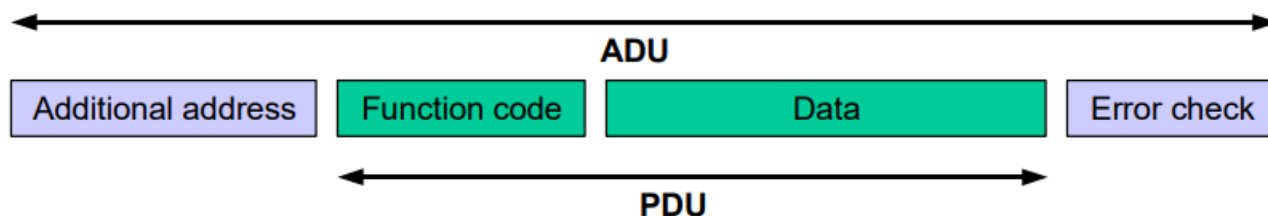
4.2. Protokol MODBUS RTU

MODBUS je industrijski protokol razvijen na aplikativnom sloju OSI modela. Zasnovan je na klijent/server (master/slave) modelu komunikacije. Na nižim slojevima može da koristi neki vid serijske veze / protokola (UART, RS-232, RS-485, ...) ili se frejm podataka aplikativnog sloja može upakovati u TCP/IP protokol i preneti putem Ethernet veze. Ukoliko se koristi serijska veza protokol se naziva MODBUS RTU, dok se u drugom slučaju naziva MODBUS TCP. Što se tiče MODBUS mreže, topologije mogu da budu hibridne, može da bude bilo koja od osnovnih topologija, odnosno više topologija zajedno [16]. Na slici 4.5 prikazan je jedan primjer mrežne topologije sa MODBUS uređajima.



Slika 4.5 – Mreža povezanih MODBUS uređaja [16]

S obzirom da je cilj projektnog zadatka implementacija MODBUS RTU klijenta (master) na strani gejtveja, nadalje će se razmatrati samo MODBUS RTU protokol (iako se opšte razmatranje protokola na aplikativnom sloju ne razlikuje). MODBUS protokol na aplikativnom sloju definiše vrlo jednostavnu jedinicu podataka protokola (*Protocol Data Unit - PDU*) koja je prikazana na slici 4.6. U zavisnosti od nižih slojeva protokola mogu se dodati još neka polja (PDU postaje *Application Data Unit - ADU*).



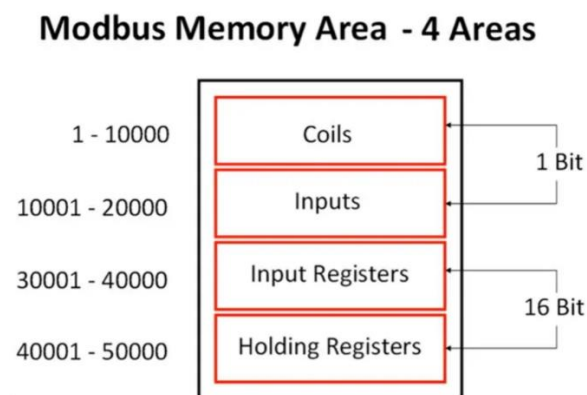
Slika 4.6 – MODBUS PDU frejm [16]

ADU se sastoji od 4 polja čiji je opis dat u nastavku [16]:

- Adresa uređaja (*Additional address*) – Adresa MODBUS uređaja kom se šalje dati ADU. Tipično veličine jedan bajt.
- Kod funkcije (*Function code*) – Osmobitni podatak koji označava radnju koja treba da se izvrši datom porukom. Ovo polje je najbitnije polje MODBUS poruke jer MODBUS slave uređaju govori šta treba da uradi kada primi poruku od mastera. Takođe slave koristi ovo polje pri odgovoru da označi da li je radnja uspješno obavljena (neuspješna radnja označena je postavljenim bitom najveće težine na ovom polju). Vrijednosti za ovo polje su definisane specifikacijom protokola, ali se mogu dodati i korisnički definisane. Treba voditi računa da se validne vrijednosti kreću od 1 do 127, dok se vrijednosti od 128 do 255 ostavljaju radi odgovora u slučaju greške.
- Podaci (*Data*) – Ovo polje sadrži konkretne podatke koji se prenose porukom. U slučaju nekih komandi, ovo polje može biti i prazno. Sadržaj polja zavisi od tipa poruke i uglavnom sadrži neku vrijednost koja treba da se upiše u memoriju slave uređaja (kada master šalje komandu), odnosno sadrži neku vrijednost iz memorije slave uređaja koju on šalje porukom do master uređaja (kada master „čita“).
- Kod greške (*Error check*) – Ovo polje je namijenjeno za detekciju greške prilikom prenosa podataka. Najčešće je to neka vrsta zbirne sume (*checksum*) ili neki od dobro poznatih metoda za detekciju greške, na primjer CRC (*Cyclic Redundancy Check*).

U MODBUS komunikaciji, master uređaj uvijek prvi započinje razmjenu. Dakle, slično kao kod IEC 101 nebalansiranog režima, master uređaj mora prvo da inicira zahtjev nakon čega slave uređaj odgovara. Ne postoji način da slave uređaj započne sam komunikaciju, odnosno svaka razmjena mora da počne od strane master uređaja. Ono što je još važno pomenuti kada se priča o MODBUS protokolu jeste način na koji se podaci slave uređaja logički prezentuju, poznato još i kao MODBUS model podataka

(MODBUS Data Model). Fizički raspored podataka u memoriji ne mora da odgovara ovom rasporedu, ali se na logičkom nivou podaci posmatraju na način prikazan na slici 4.7.



Slika 4.7 – Memorijski model podataka MODBUS protokola [17]

Model podataka sastoji se od četiri dijela čiji je opis dat u nastavku [17]:

- Diskretni izlazi (*Coils*) – Tip podataka koji predstavlja izlazni element koji može imati dva stanja – uključen i isključen (npr. relej). Adresni prostor ovog dijela modela čine adrese od 1 do 10000. Za predstavljanje ovog podatka dovoljan je jedan bit, ali se tipično u adresnom prostoru koristi jedan bajt. Podatak ovog tipa je dostupan master uređaju i za čitanje i upis.
- Diskretni ulazi (*Discrete Inputs*) - Tip podataka koji predstavlja ulazni element koji može imati dva stanja – uključen i isključen (npr. prekidač). Adresni prostor ovog dijela modela čine adrese od 10001 do 20000. Za predstavljanje ovog podatka dovoljan je jedan bit, ali se tipično u adresnom prostoru koristi jedan bajt. Podatak ovog tipa je dostupan master uređaju samo za čitanje.
- Ulazni registri (*Input Registers*) – Tip podatka koji služi za skladištenje ulaznih vrijednosti koje su predstavljene dvobajtnim podatkom (npr. mjerenja dobijena sa Analogno/Digitalnog konvertora). Adresni prostor ovog dijela modela čine adrese od 30001 do 40000. Podatak ovog tipa je dostupan master uređaju samo za čitanje.
- Izlazni registri (*Holding Registers*) – Tip podatka koji služi za skladištenje izlaznih vrijednosti koje su predstavljene dvobajtnim podatkom (npr. vrijednosti koje treba pretvoriti Digitalno/Analognim konverterom). Adresni prostor ovog dijela modela čine adrese od 40001 do 50000. Podatak ovog tipa je dostupan master uređaju i za čitanje i za upis.

Razumijevanje modela podataka, tipova podataka, vrsta komandi, zahtjeva i poruka je neophodno za jedan ključan korak pri projektovanju aplikacije gejtveja, a to je mapiranje između protokola. Ovaj korak će biti posebno obrađen u nekoj od narednih podglava. Ovim izlaganjem ukratko je predstavljen MODBUS protokol što je neophodno da bi se razumjela osnovna terminologija kao i rad sa korišćenom bibliotekom za implementaciju MODBUS master uređaja koji će se izvršavati na gejtveju.

4.2.1. MODBUS biblioteka – *libmodbus*

Biblioteka *libmodbus* je najpoznatija biblioteka otvorenog koda koja nudi implementaciju MODBUS protokola (i RTU i TCP) u C programskom jeziku. Biblioteka nudi API funkcije za implementaciju i slave i master uređaja. Predstavlja vrlo dobar izbor za ugrađene sisteme jer dolazi bez dodatnih zavisnosti i može se smatrati za *lightweight* biblioteku [18]. Sam API biblioteke je vrlo jednostavan, ne sadrži previše funkcija i poprilično je lak za korišćenje i dobro dokumentovan. S obzirom da je cilj projektnog zadatka implementacija MODBUS RTU master uređaja, u nastavku će biti spomenute osnovne funkcije koje su neophodne za željenu implementaciju. Prvi korak ka implementaciji jeste da se kreira odgovarajuća instanca MODBUS konteksta (*modbus_t* tipa) pozivom funkcije *modbus_new_rtu*. Ova funkcija očekuje naziv serijskog porta koji se koristi za realizaciju serijske veze, kao i odgovarajuće parametre te serijske veze (bitska brzina, broj start i stop bita, paritet). Ukoliko je kreiranje uspješno, vraća se instanca novog RTU konteksta koji sada može da se koristi za slanje odgovarajućih poruka (ADU). Prije nego što je slanje omogućeno, potrebno je pozvati funkciju koja će master uređaj povezati na serijski port – *modbus_connect*. Ako je konekcija uspješna, ostvareni su svi uslovi za razmjenu poruka. Svaka razmjena poruka u implementaciji MODBUS mastera započinje specificiranjem adrese slave uređaja (polje *Additional address*) pozivom funkcije *modbus_set_slave*. Ukoliko postoji samo jedan slave uređaj, ova funkcija se može pozvati samo jednom na početku. U suprotnom, kada god se mijenja slave uređaj sa kojim se komunicira, potrebno je pozvati ovu funkciju. Nakon „odabira“ slave uređaja, konačno se može poslati zahtjev koji može biti čitanje (poziv funkcije tipa *modbus_read_**) ili upis (poziv funkcije tipa *modbus_write_**). Pored ovih glavnih funkcija, API nudi neke dodatne funkcije za podešavanje parametara komunikacije, podešavanje vremena čekanja na odgovor (*timeout*), funkcije nižeg nivoa za manipulaciju serijskom vezom i ostale. Svi detalji oko ovih funkcija i samog API-ja mogu se pronaći na sajtu zvanične dokumentacije [18].

4.3. Mapiranje između IEC 104 i MODBUS protokola

U ovoj podglavi biće predstavljen način na koji je izvršeno mapiranje između dva protokola. Ovo mapiranje podrazumijeva uspostavljanje ekvivalencije između osnovnih tipova podataka dva protokola i između osnovnih poruka (komandi) koji se koriste u oba protokola. Ovo je glavni korak pri projektovanju aplikacije gejtveja, jer se pronađena mapiranja direktno prevode u implementaciju i olakšavaju taj proces. U tabeli 4.1 prikazana su mapiranja između osnovnih IEC 104 tipova podataka i MODBUS tipova podataka. Pored mapiranja osnovnih tipova podataka, još je neophodno mapirati i osnovne zahtjeve i komande koje gejtvej može primiti od IEC 104 kontrolišuće stanice (master stanice) u zahtjeve i komande koji postoje u MODBUS protokolu. Ovo mapiranje dato je u tabeli 4.2. Bitno je napomenuti da u trenutnoj implementaciji gejtveja nije izvršena kompletna implementacija ovih mapiranja već samo određena koja zadovoljavaju glavne funkcionalnosti. Svakako, bitno je prikazati i projektovati kompletno mapiranje, jer je kasnije u slučaju potrebe za implementacijom to mnogo lakše izvesti. Detalji implementacije biće prikazani u nekoj od narednih podglava.

Tabela 4.1 – Mapiranje osnovnih tipova podataka

IEC 104 tip podatka	MODBUS tip podatka
Jedinična tačka (<i>Single point</i>)	Diskretni izlaz / ulaz (<i>Coil / Discrete input</i>)
Dvostruka tačka (<i>Double point</i>)	2 x Diskretni izlaz / ulaz (<i>Coil / Discrete input</i>)
Normalizovana vrijednost (<i>Normalized value</i>)	Ulazni / izlazni registar (<i>Input / Holding register</i>)
Skalirana vrijednost (<i>Scaled value</i>)	Ulazni / izlazni registar (<i>Input / Holding register</i>)
Realan broj jednostruke preciznosti (<i>Short float</i>)	2 x Ulazni / izlazni registar (<i>Input / Holding register</i>)
Vrijednost binarnog brojača (<i>Binary counter reading</i>)	2 x Ulazni / izlazni registar (<i>Input / Holding register</i>)
Vrijednost sa indikatorom tranzijenta (<i>Value with transient state indication</i>)	Ulazni / izlazni registar (<i>Input / Holding register</i>)

Tabela 4.2 – Mapiranje zahtjeva i komandi

IEC 104 zahtjev / komanda	MODBUS implementacija zahtjeva / komande
Zahtjev za interogaciju (<i>Interrogation request</i>)	Čitanje svih ulaznih i izlaznih registara kao i diskretnih ulaza i izlaza ciljnog slave uređaja
Jedinična komanda (<i>Single command</i>)	Promjena stanja određenog diskretnog izlaza ciljnog slave uređaja
Dvostruka komanda (<i>Double command</i>)	Promjena stanja para diskretnih izlaza ciljnog slave uređaja
Koračna komanda (<i>Regulating step command</i>)	Inkrementovanje / dekrementovanje vrijednosti određenog izlaznog registra ciljnog slave uređaja
Komanda za postavljanje skalirane / normalizovane vrijednosti (<i>Setpoint command, scaled / normalized value</i>)	Postavljanje vrijednosti određenog izlaznog registra ciljnog slave uređaja
Komanda za postavljanje vrijednosti realnog broja / niza bita (<i>Setpoint command, short FP value / bistring</i>)	Postavljanje vrijednosti para izlaznih registra ciljnog slave uređaja
Komanda za čitanje (<i>Read command</i>)	Čitanje specificirane vrijednost ciljnog slave uređaja koja može biti i diskretni ulaz / izlaz i ulazni / izlazni registar u zavisnosti od adrese info. objekta

4.4. Konfiguracija *gateway* uređaja

Prije prikaza detalja implementacije gejtveja, još je bitno razjasniti jednu temu, a to je način konfiguracije samog gejtveja. Da bi gejtvej mogao ispravno funkcionisati potrebno je da informacije o povezanim uređajima budu poznate prije pokretanja aplikacije gejtveja. S obzirom da je gejtvej povezan sa jednim IEC 104 master uređajem (SCADA PC) to znači da je nepoznata konfiguracija sa strane MODBUS dijela. Drugim riječima, gejtveju je neophodno na neki način, prije pokretanja aplikacije, omogućiti odgovarajuću konfiguraciju u kojoj će biti navedeni svi MODBUS slave uređaji povezani na serijske portove gejtveja. Bez ovog konfiguracionog mehanizma, gejtvej bi morao da proziva MODBUS slave uređaje što je izuzetno vremenski zahtjevno i neefikasno. Takođe, ovo prozivanje uređaja bi samo dalo informaciju o povezanim uređajima i njihovim adresama bez informacije o postojećim registrima, ulazima i izlazima koje ima dati slave uređaj. MODBUS protokol ne nudi neki poseban mehanizam kojim bi mogla da se dobije memorijska slika svih registara slave uređaja na jednostavan način, tako da je jedini ispravan put za dobijanje informacije o slave uređajima kreiranje odgovarajuće konfiguracije prije vremena izvršavanja. Jedan od najpoznatiji načina implementacije konfiguracija u softverskim sistemima jeste korišćenje fajlova odgovarajućih formata. U slučaju ovog projektnog zadatka, kompletna konfiguracija smješta se u jedan JSON (*JavaScript Object Notation*) fajl. JSON format je *lightweight* format veoma popularan za razmjenu podataka preko odgovarajućih konfiguracionih fajlova. Najveća prednost mu je jednostavnost, kako za čitanje i pisanje čovjeku, tako i za parsiranje i generisanje putem mašine (računara). JSON fajlovi sastoje se od dvije fundamentalne strukture: kolekcije objekata parova „naziv-vrijednost“ i uređene liste vrijednosti (nizova). Svaki objekat JSON fajla počinje i završava se vitičastim zagradama „{ }“. Unutar vitičastih zagrada dodaju se atributi objekta u formi neke od dvije navedene osnovne strukture [19].

Konfiguracioni fajl implementiran za ovaj projektni zadatak sastoji se od jednog glavnog objekta, niza naziva *port*. Ovaj niz sastoji se od više objekata koji predstavljaju konfiguraciju određenog serijskog porta. Svaki od tih objekata sastoji se od sledećih atributa:

- *value* – Redni broj serijskog porta gejtveja (u Linux sistemima to je broj X koji se dobije iz /dev foldera za serijske uređaje *ttySX*).
- *active* – Označava da li je serijski port aktivan, odnosno da li na njemu ima povezanih slave uređaja. Očekivane vrijednosti su 0 za neaktivan i 1 za aktivan port.
- *baud_rate* – Konfiguracioni parametar za bitsku brzinu serijskog porta. Očekivane vrijednosti su neke od standardnih vrijednosti za bitske brzine serijskog UART protokola (npr. 9600, 19200, 115200, ...).
- *data_bits* – Konfiguracioni parametar za broj bita podataka serijskog porta. Najčešće je to 8 bita, ali su podržane vrijednosti i 4, 5, 6 i 7.
- *stop_bits* – Konfiguracioni parametar za broj stop bita serijskog porta. Očekivana vrijednost je 1 ili 2.

- *parity* – Konfiguracioni parametar koji označava tip bita parnosti (ako se koristi). Očekivane vrijednosti su 0 ako nema bita parnosti, 1 za bit neparnog (*odd*) pariteta i 2 za bit parnog (*even*) pariteta.
- *slaves* – Niz objekata koji predstavljaju MODBUS slave uređaje povezane na dati serijski port.

S obzirom da gejtvej uređaj koristi šest serijskih portova, u konfiguracioni fajl neophodno je dodati SVIH šest objekata u niz *port*, čak i u slučaju da nema povezanih slave uređaja na nekim od njih (u tom slučaju za konkretan port *active* fleg se postavlja na 0, a *slaves* niz se ostavlja praznim). Na slici 4.8 prikazan je primjer definisanja *port* niza sa jednim aktivnim i dva neaktivna port objekta.

```
{
  "port":
  [
    {
      "value": 1,
      "active": 0,
      "baud_rate": 0,
      "data_bits": 0,
      "stop_bits": 0,
      "parity": 0,
      "slaves":
      [
      ]
    },
    {
      "value": 2,
      "active": 0,
      "baud_rate": 0,
      "data_bits": 0,
      "stop_bits": 0,
      "parity": 0,
      "slaves":
      [
      ]
    },
    {
      "value": 3,
      "active": 1,
      "baud_rate": 19200,
      "data_bits": 8,
      "stop_bits": 2,
      "parity": 0,
      "slaves":
      [
      ]
    }
  ]
}
```

Slika 4.8 – JSON port niz sa 3 objekta

Kao što je opisano, svaki objekat u *port* nizu sastoji se od atributa *slaves* koji je takođe niz objekata koji reprezentuju jedan MODBUS slave uređaj. Ovi objekti takođe imaju svoje atribute neophodne za konfiguraciju gejtveja, odnosno gejtvej preko ovih objekata pamti konkretne slave uređaje, na koji serijski port su povezani, koje registre, ulaze i izlaze posjeduju kao i neke dodatne parametre. Svaki objekat niza *slaves* sastoji se od sledećih atributa:

- *id* – Identifikator (adresa) MODBUS slave uređaja.

- *description* – Opis (naziv) konkretnog slave uređaja. Nema ulogu za samu implementaciju, ali može pomoći kao vid dokumentacije ili za lakše informisanje i razumijevanje.
- *coils* – Niz objekata koji predstavljaju diskretne izlaze koje posjeduje konkretni slave uređaj. Niz se sastoji od parova naziv-vrijednost pri čemu svaka vrijednost predstavlja adresu diskretnog izlaza unutar memorije konkretnog slave uređaja.
- *discrete_inputs* – Niz objekata koji predstavljaju diskretne ulaze koje posjeduje konkretni slave uređaj. Niz se sastoji od parova naziv-vrijednost pri čemu svaka vrijednost predstavlja adresu diskretnog ulaza unutar memorije konkretnog slave uređaja.
- *input_registers* – Niz objekata koji predstavljaju ulazne registre koje posjeduje konkretni slave uređaj. Niz se sastoji od parova naziv-vrijednost pri čemu svaka vrijednost predstavlja adresu ulaznog registra unutar memorije konkretnog slave uređaja.
- *holding_registers* – Niz objekata koji predstavljaju izlazne registre koje posjeduje konkretni slave uređaj. Niz se sastoji od parova naziv-vrijednost pri čemu svaka vrijednost predstavlja adresu izlaznog registra unutar memorije konkretnog slave uređaja.

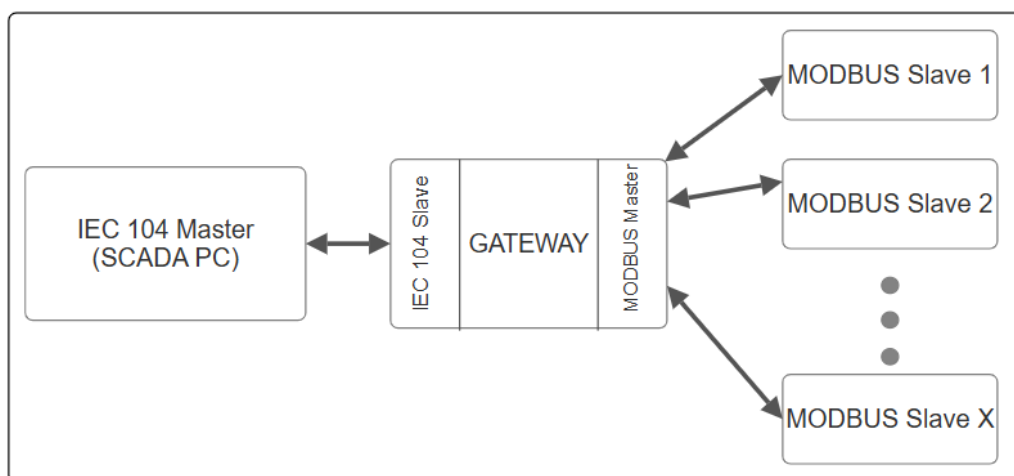
Ovi atributi dovoljni su da opišu sve osnovne informacije koje su potrebne za konfiguraciju gejtveja. Iako se čini da je konfiguracija vezana samo za MODBUS protokol i serijsku vezu, ona se zapravo vrlo jednostavno primjenjuje i kao konfiguracija za IEC 104 stranu gejtveja primjenjivanjem odgovarajućih mapiranja između podataka opisanim u podglavi 4.3. Primjer dodavanja konfiguracije jednog slave uređaja koji se nalazi u objektu niza *port* sa vrijednošću 3, dat je na slici 4.9.

```
{
  "id": 35,
  "description": "Temperature sensor",
  "coils":
  [
    {"address": 0},
    {"address": 1},
    {"address": 2}
  ],
  "discrete_inputs":
  [
    {"address": 0},
    {"address": 1},
    {"address": 2}
  ],
  "input_registers":
  [
    {"address": 0},
    {"address": 1},
    {"address": 2}
  ],
  "holding_registers":
  [
    {"address": 0},
    {"address": 1},
    {"address": 2}
  ]
},
```

Slika 4.9 – Konfiguracija jednog objekta u nizu slaves

4.5 Softverska implementacija sistema

Softverska implementacija sistema podrazumijeva pisanje koda u odgovarajućem programskom jeziku u cilju razvoja izvršnog programa koji se može pokrenuti na Linux operativnom sistemu i koji će implementirati zahtjeve konkretnog gejtvej uređaja. Kompletan projekat i implementacija dostupni su na GitHub platformi [20]. Za implementaciju korišten je programski jezik C uz standardnu biblioteku *glibc* i dodatne tri *third party* biblioteke od kojih su dvije već spomenute, *lib60870* i *libmodbus*. Treća korištena biblioteka je biblioteka *jansson* koja se koristi u svrhu parsiranja JSON konfiguracionog fajla. Ova biblioteka je jednostavna *lightweight* C biblioteka za parsiranje i kreiranje JSON fajlova preporučljiva za ugrađene računarske sisteme [21]. API biblioteke je vrlo jednostavan i u kontekstu ovog projektnog zadatka dovoljne su samo funkcionalnosti za osnovno parsiranje JSON fajla. Da bi se bolje razumjela implementacija gejtveja, na slici 4.10 prikazana je blok šema proizvoljnog sistema u kom gejtvej može da se nađe.



Slika 4.10 – Blok šema sistema sa gejtvej uređajem

Sa prethodne blok šeme je očigledno da gejtvej uređaj treba sa jedne strane da implementira IEC 104 slave uređaj, a sa druge strane MODBUS master uređaj. Ideja prilikom implementacije je bila da se prvo implementira strana MODBUS mastera, na takav način da implementacija obezbjedi odgovarajuće funkcije (API) tako da se lako može integrisati sa IEC 104, odnosno, u suštini, sa bilo kojim drugim protokolom. Nakon toga, u izvornom kodu sa konkretnom aplikacijom bi se korišćenjem datog API-ja jednostavno izvršila implementacija IEC 104 slave strane te primjenila odgovarajuća mapiranja (diskutovana u podglavi 4.3). Ova mapiranja bi jednostavno dovela do direktnih poziva funkcija MODBUS master API-ja koje bi obavile traženi zahtjev, nakon čega se MODBUS odgovor takođe mapira i vraća nazad IEC 104 master uređaju. Dakle, prvi korak jeste implementacija MODBUS master funkcionalnosti koji odgovaraju zahtjevima i koji se mogu lako integrisati u veću aplikaciju za mapiranje sa drugim protokolima. Ova implementacija će biti prikazana u nastavku.

4.5.1. Implementacija MODBUS mastera

Implementacija MODBUS mastera naslanja se na spomenutu biblioteku *libmodbus* i u suštini predstavlja jedan sloj između konkretne aplikacije gejtveja i ove biblioteke koji apstrahuje pozive funkcija biblioteke i olakšava ih na način da se lako pozivaju u kontekstu gejtveja i konkretnih mapiranja. S obzirom da gejtvej može da ima veliki broj slave uređaja koji su povezani na različite serijske portove, neophodan je odgovarajući interfejs koji će da pronađe željeni uređaj na odgovarajućem serijskom portu te da izvrši datu komandu. Dakle, korišćenje samih funkcija biblioteke *libmodbus* direktno iz aplikacije gejtveja prethodi dodatnoj obradi, translaciji adresa i indeksiranju konkretnog slave uređaja, što znači da je implementacija ovakvih funkcionalnosti mnogo bolja u zasebnom dijelu koda. Pored glavnih funkcionalnosti za čitanje / pisanje podataka, implementacija MODBUS mastera sadrži i druge funkcionalnosti neophodne za rad kompletne aplikacije. U njoj se nalaze funkcionalnosti za parsiranje konfiguracionog fajla, kreiranje modbus konteksta za sve korištene serijske portove, kao i dodatne funkcije za debugovanje i oslobađanje zauzetih memorijskih resursa. Spisak svih dostupnih funkcija koje su implementirane u sklopu MODBUS mastera dat je na slici 4.11.

Functions	
<code>uint8_t*</code>	<code>parse_address_array (json_t *json_array, uint8_t *count)</code> Function that parses slave configuration of json config file for slave memory layout.
<code>simple_slave_t**</code>	<code>parse_slaves (json_t *root, uint8_t *num_of_slaves, serial_configuration_t *cfg)</code> Function that parses the json config file and searches for slave devices configuration.
<code>void</code>	<code>free_slaves (simple_slave_t **slaves, uint8_t *num_of_slaves)</code> Function that releases the memory allocated for slave device objects.
<code>void</code>	<code>free_modbus (modbus_t **ctx)</code> Function that releases memory used by modbus context objects and modbus connections.
<code>void</code>	<code>free_interrogation_response (interrogation_response_t *resp)</code> Function that releases memory used to store interrogation response data.
<code>simple_slave_t**</code>	<code>init_slaves (const char *cfg_file, uint8_t *num_of_slaves, serial_configuration_t *cfg)</code> Function that initializes arrays of active slave devices derived from configuration file in json format.
<code>modbus_t*</code>	<code>init_modbus_connection (const char *dev_path, uint32_t baud, uint8_t parity, uint8_t data_bits, uint8_t stop_bits)</code> Function that initializes modbus connection and creates a modbus context object.
<code>void</code>	<code>print_slaves (simple_slave_t **slaves, uint8_t *num_of_slaves)</code> Function to print the information about slave devices found in config file.
<code>uint8_t</code>	<code>get_slave_idx (uint16_t slave_id, simple_slave_t *slaves, uint8_t num_of_slaves)</code> Function that retrieves index of a slave with the given ID from the slave array.
<code>interrogation_response_t*</code>	<code>interrogate_slave (uint16_t slave_id, simple_slave_t *slaves, uint8_t num_of_slaves, modbus_t *ctx)</code> Function that gathers data of all coils, inputs and registers of the specified slave.
<code>uint8_t*</code>	<code>read_coil (uint16_t slave_id, uint8_t coil_addr, simple_slave_t *slaves, uint8_t num_of_slaves, modbus_t *ctx)</code> Function that reads status of one coil.
<code>uint8_t*</code>	<code>read_discrete_input (uint16_t slave_id, uint8_t discrete_input_addr, simple_slave_t *slaves, uint8_t num_of_slaves, modbus_t *ctx)</code> Function that reads status of one discrete input.
<code>uint16_t*</code>	<code>read_input_register (uint16_t slave_id, uint8_t input_reg_addr, simple_slave_t *slaves, uint8_t num_of_slaves, modbus_t *ctx)</code> Function that reads value from one input register.
<code>uint16_t*</code>	<code>read_holding_register (uint16_t slave_id, uint8_t holding_reg_addr, simple_slave_t *slaves, uint8_t num_of_slaves, modbus_t *ctx)</code> Function that reads value from one holding register.
<code>uint8_t</code>	<code>write_coil (uint16_t slave_id, uint8_t coil_addr, uint8_t coil_value, simple_slave_t *slaves, uint8_t num_of_slaves, modbus_t *ctx)</code> Function that sets state of one coil.
<code>uint8_t</code>	<code>write_holding_register (uint16_t slave_id, uint8_t holding_reg_addr, uint16_t holding_reg_value, simple_slave_t *slaves, uint8_t num_of_slaves, modbus_t *ctx)</code> Function that writes value to one holding register.

Slika 4.11 – MODBUS master API

Prva funkcija od značaja je funkcija *init_slaves*. Ova funkcija služi za parsiranje konfiguracionog JSON fajla odakle dobija informacije o povezanim slave uređajima na serijskim portovima gejtveja. Nakon uspješnog izvršenja vraća ukupno šest nizova (koliko postoji serijskih portova na ciljnoj platformi) koji sadrže informacije o povezanim slave uređajima (struktura *simple_slave*), tako da svaki indeks niza (uvećan za jedan) odgovara serijskom portu na koji su slave uređaji povezani. Takođe funkcija popunjava i niz *num_of_slaves* koji sadrži informaciju o broju povezanih slave uređaja na konkretnom portu te niz strukture *serial_configuration* koja sadrži parametre serijske veze svakog od portova. Implementacija ove funkcije data je na slici 4.12.

```
simple_slave_t** init_slaves(const char* cfg_file, uint8_t* num_of_slaves, serial_configuration_t* cfg)
{
    simple_slave_t** slaves = NULL;
    json_error_t error;

    json_t* root = json_load_file(cfg_file, 0, &error);

    if(root == NULL)
    {
        #ifdef PRINT_DEBUG
        fprintf(stderr, "Error parsing JSON: %s (line %d, column %d)\n", error.text, error.line, error.column);
        #endif
        return NULL;
    }

    slaves = parse_slaves(root, num_of_slaves, cfg);
    json_decref(root);

    return slaves;
}
```

Slika 4.12 – Implementacija funkcije *init_slaves*

Za svoj rad ova funkcija poziva funkciju *parse_slaves*, koja koristi funkciju *parse_address_array* da bi se na kraju dobila kompletna mapa svih slave uređaja i njihovih registara. Ono što je ovdje jako bitno napomenuti jeste da na dva različita serijska porta u opštem slučaju mogu biti povezani slave uređaji koji imaju iste adrese. Zbog toga se unutar implementacije prilikom parsiranja slave uređaja, na svaku adresu pojedinog uređaja dodaje odgovarajući *offset* koji je jednak $n * 1000$, gdje je n redni broj porta na koji je povezan dati slave uređaj. Stvarna adresa slave uređaja se dobija kada se dati *offset* oduzme, i ova translacija mora da se uradi u svakoj funkciji koja treba da razmjeni podatke sa nekim slave uređajem. Na slikama 4.13, 4.14 i 4.15 prikazani su dijelovi koda od interesa iz prethodno spomenutih funkcija. Na ovim slikama kao i na prethodnoj slici, može se takođe vidjeti način upotrebe pomenute biblioteke za rad sa JSON fajlovima, biblioteke *jansson*.

```
port_obj = json_array_get(port_array, j);
active = (uint8_t) json_integer_value(json_object_get(port_obj, "active"));
port_value = (uint8_t) json_integer_value(json_object_get(port_obj, "value"));
cfg[j].baud_rate = (uint32_t) json_integer_value(json_object_get(port_obj, "baud_rate"));
cfg[j].data_bits = (uint8_t) json_integer_value(json_object_get(port_obj, "data_bits"));
cfg[j].stop_bits = (uint8_t) json_integer_value(json_object_get(port_obj, "stop_bits"));
parity_tmp = (uint8_t) json_integer_value(json_object_get(port_obj, "parity"));

if(parity_tmp == 0)
{
    cfg[j].parity = MODBUS_PARITY_NONE;
}
else if(parity_tmp == 1)
{
    cfg[j].parity = MODBUS_PARITY_ODD;
}
else
{
    cfg[j].parity = MODBUS_PARITY_EVEN;
}
```

Slika 4.13 – Funkcija `parse_slaves`, parsiranje parametara serijske veze

```
slave_obj = json_array_get(slaves_array, i);

// Parse ID and description
slaves[j][i].id = port_value * OFFSET_BY_PORT + (uint8_t)json_integer_value(json_object_get(slave_obj, "id"));
strncpy(slaves[j][i].name, json_string_value(json_object_get(slave_obj, "description")), MAX_SLAVE_NAME_LEN);

// Parse coils addresses
json_t* coils_array = json_object_get(slave_obj, "coils");
slaves[j][i].coils_addr = parse_address_array(coils_array, &slaves[j][i].num_of_coils);

// Parse discrete inputs addresses
json_t* discrete_inputs_array = json_object_get(slave_obj, "discrete_inputs");
slaves[j][i].discrete_inputs_addr = parse_address_array(discrete_inputs_array, &slaves[j][i].num_of_discrete_inputs);

// Parse input registers addresses
json_t* input_registers_array = json_object_get(slave_obj, "input_registers");
slaves[j][i].input_registers_addr = parse_address_array(input_registers_array, &slaves[j][i].num_of_input_registers);

// Parse holding registers addresses
json_t* holding_registers_array = json_object_get(slave_obj, "holding_registers");
slaves[j][i].holding_registers_addr = parse_address_array(holding_registers_array, &slaves[j][i].num_of_holding_registers);
```

Slika 4.14 – Funkcija `parse_slaves`, parsiranje pojedinačnih slave objekata

```
uint8_t* parse_address_array(json_t* json_array, uint8_t* count)
{
    uint8_t size = json_array_size(json_array);
    uint8_t* addresses = (uint8_t*)malloc(size * sizeof(uint8_t));

    if (addresses == NULL)
    {
        #ifdef PRINT_DEBUG
        fprintf(stderr, "Failed to allocate memory for address array !\n");
        #endif
        return NULL;
    }

    for (uint8_t i = 0; i < size; i++)
    {
        json_t* item = json_array_get(json_array, i);
        json_t* address_obj = json_object_get(item, "address");
        addresses[i] = (uint8_t)json_integer_value(address_obj);
    }

    *count = (uint8_t)size;
    return addresses;
}
```

Slika 4.15 – Implementacija funkcije `parse_address_array`

Nakon uspješnog parsiranja konfiguracionog fajla, potrebno je inicijalizovati MODBUS kontekste na svim aktivnim serijskim portovima, što obavlja poziv funkcije *init_modbus_connection* za svaki pojedinačan port. Implementacija ove funkcije data je na slici 4.16.

```
modbus_t* init_modbus_connection(const char* dev_path, uint32_t baud, uint8_t parity, uint8_t data_bits, uint8_t stop_bits)
{
    modbus_t* ctx = modbus_new_rtu(dev_path, baud, parity, data_bits, stop_bits);
    if (ctx == NULL)
    {
        #ifdef PRINT_DEBUG
            fprintf(stderr, "Unable to create the libmodbus context: %s\n", modbus_strerror(errno));
        #endif
        return NULL;
    }

    /* Set RS485 serial mode */
    /*
    rc = modbus_rtu_set_serial_mode(ctx, MODBUS_RTU_RS485);
    if(rc == -1)
    {
        fprintf(stderr, "Failed to set RS485 serial mode: %s\n", modbus_strerror(errno));
        return NULL;
    }
    */

    /* Set timeout */
    /*
    modbus_set_byte_timeout(ctx, 0, 0);
    modbus_set_response_timeout(ctx, 0, RESPONSE_TIMEOUT);
    */

    /* Connect to the line */
    if(modbus_connect(ctx) == -1)
    {
        #ifdef PRINT_DEBUG
            fprintf(stderr, "Modbus connection failed: %s\n", modbus_strerror(errno));
        #endif
        return NULL;
    }

    return ctx;
}
```

Slika 4.16 – Implementacija funkcije *init_modbus_connection*

Funkcije koje počinju sa *read* ili *write* su konkretne funkcije koje se koriste za razmjenu podataka sa MODBUS slave uređajem. Ove funkcije vrše odgovarajuće translacije slave adrese i koriste objekat MODBUS konteksta da pozovu odgovarajuću funkciju iz *libmodbus* biblioteke koja će da izvrši željeni zahtjev. Primjer jedne takve funkcije dat je na slici 4.17. Postoji još jedna specijalna funkcija *interrogate_slave* posebno kreirana u svrhu korišćenja sa IEC 104 protokolom radi obezbjeđivanja gotovog mehanizma u slučaju zahtjeva za interogacijom, koji je najčešći zahtjev koji dolazi od strane IEC 104 master uređaja. Ova funkcija jednostavno čita sadržaje svih registara, ulaza i izlaza željenog slave uređaja i rezultate smješta u posebnu strukturu podataka *interrogation_response* koja se kasnije može koristiti za mapiranje rezultata i vraćanje IEC 104 master uređaju. Funkcije koje počinju sa *free* služe za oslobađanje memorijskih resursa, a postoje i dodatne funkcije u svrhe debugovanja, kao što je *print_slaves* (da se provjeri da li je parsiranje konfiguracionog fajla bilo uspješno). Pored opisanih funkcija i struktura podataka, API sadrži dodatne konstante koje se mogu koristiti prilikom rada sa funkcijama.

```

uint8_t* read_coil(uint16_t slave_id, uint8_t coil_addr, simple_slave_t* slaves, uint8_t num_of_slaves, modbus_t* ctx)
{
    uint8_t idx = 0;
    uint8_t* res = NULL;
    uint8_t real_slave_id = (uint8_t)(slave_id - ((uint16_t)(slave_id / OFFSET_BY_PORT)) * OFFSET_BY_PORT);

    if(slaves == NULL)
    {
        #ifdef PRINT_DEBUG
            fprintf(stderr, "Failed to read coil status, slave object is NULL.\n");
        #endif
        return NULL;
    }

    idx = get_slave_idx(slave_id, slaves, num_of_slaves);

    if(idx >= num_of_slaves)
    {
        #ifdef PRINT_DEBUG
            fprintf(stderr, "Failed to read coil status, invalid slave ID.\n");
        #endif
        return NULL;
    }

    modbus_set_slave(ctx, real_slave_id);
    for(uint8_t i = 0; i < slaves[idx].num_of_coils; i++)
    {
        if(slaves[idx].coils_addr[i] == coil_addr)
        {
            res = (uint8_t*) calloc(1, sizeof(uint8_t));
            modbus_read_bits(ctx, coil_addr, 1, res);

            #ifdef PRINT_DEBUG
                fprintf(stdout, "Coil address: %u, status: %s\n", coil_addr, *res ? "ON" : "OFF");
            #endif
            break;
        }
    }

    #ifdef PRINT_DEBUG
        if(res == NULL)
        {
            fprintf(stderr, "Failed to read coil status, invalid coil address.\n");
        }
    #endif

    return res;
}

```

Slika 4.17 – Primjer implementacije funkcije za čitanje stanja diskretnog izlaza

Kompletna implementacija i dokumentacija dostupna je na GitHub stranici projekta [20]. S obzirom da je čitav API MODBUS master strane aplikacije dokumentovan u skladu sa *Doxygen* komentarima, dokumentacija se može vrlo jednostavno generisati *Doxygen* alatom. Takođe se može pronaći i implementacija testne aplikacije koja testira samo funkcionalnosti implementiranog MODBUS mastera (*test_app.c*).

4.5.2. Integracija MODBUS mastera i implementacija aplikacije gejtveja

Nakon implementacije odgovarajućeg API za MODBUS master dio gejtveja, može se pristupiti narednom koraku, a to je integracija sa IEC 104 slave dijelom i kompletiranje aplikacije. Kompletan izvorni kod aplikacije može se naći na GitHub stranici projekta, u fajlu *simple_server.c*. Glavnu ulogu u implementaciji IEC 104 slave dijela ima pomenuta biblioteka *lib60870*. U tački 4.2.1. pojašnjena je osnovna logika implementacije IEC 104 slave uređaja gdje je rečeno da se sva interakcija sa komandama i zahtjevima od strane IEC 104 master uređaja obavlja putem implementacije odgovarajućih *callback handler* funkcija. Implementirane *callback* funkcije se potom registruju pozivom odgovarajućih funkcija

iz *lib60870* biblioteke, ali prije toga je potrebno izvršiti kreiranje IEC 104 slave objekta i njegovu inicijalizaciju. Na slici 4.18 prikazan je dio inicijalizacionog koda gejtvej aplikacije.

```
/* Prepare variables for modbus master initialization */
int rc = 0;
serial_configuration_t cfg[SERIAL_PORTS_NUM];
modbus_communication_param_t mb_comm_param;

/* Add Ctrl-C handler */
signal(SIGINT, sigint_handler);

/* Initialize modbus slaves and connections */
mb_comm_param.slaves = init_slaves(CONFIG_FILE_PATH, mb_comm_param.num_of_slaves, cfg);
if(mb_comm_param.slaves == NULL)
{
    fprintf(stderr, "Unable to get slave devices configuration.\n");
    return 0;
}

for(uint8_t i = 0; i < SERIAL_PORTS_NUM; i++)
{
    if(mb_comm_param.slaves[i] != NULL)
    {
        mb_comm_param.ctx[i] = init_modbus_connection(DEVICE_PATHS[i], cfg[i].baud_rate, cfg[i].parity, cfg[i].data_bits, cfg[i].stop_bits);
    }
    else
    {
        mb_comm_param.ctx[i] = NULL;
    }
}

print_slaves(mb_comm_param.slaves, mb_comm_param.num_of_slaves);

/* create a new slave/server instance with default connection parameters and
 * default message queue size */
CS104_Slave slave = CS104_Slave_create(10, 10);

CS104_Slave_setLocalAddress(slave, "0.0.0.0");

/* Set mode to a single redundancy group
 * NOTE: library has to be compiled with CONFIG_CS104_SUPPORT_SERVER_MODE_SINGLE_REDUNDANCY_GROUP enabled (=1)
 */
CS104_Slave_setServerMode(slave, CS104_MODE_SINGLE_REDUNDANCY_GROUP);
```

Slika 4.18 – Inicijalizacija gejtvej aplikacije

U inicijalizacionom dijelu koda javlja se nova struktura podataka, *modbus_communication_param*. Ova struktura podataka služi za skladištenje svih informacija o pročitanim MODBUS slave uređajima, njihovom broju i objektima MODBUS konteksta. Svaki *handler* koji mora da vrši čitanje / upis podataka treba da ima pristup ovim podacima. Da bi se izbjeglo korišćenje globalnih varijabli, može se iskoristiti mehanizam prosljeđivanja parametara preko pokazivača prilikom registracije *handler* funkcije, što će biti prikazano na slici 4.19. Takođe se sa slike 4.18 vidi korišćenje MODBUS master API funkcija za inicijalizaciju, koje popunjavaju podatke iz prethodno pomenute strukture. Nakon inicijalizacije MODBUS dijela, kreira se IEC 104 slave objekat pozivom funkcije *CS104_Slave_create*. Nakon toga se podešava lokalna IP adresa kreiranog slave objekta na 0.0.0.0 što znači da se sa slave uređajem može povezati bilo koji IEC 104 master uređaj u lokalnoj mreži (koja je podešena u okviru Linux OS koji se izvršava na ciljnoj platformi). Poslednja linija inicijalizacionog koda, poziv funkcije *CS104_Slave_setServerMode* podešava parametar povezivanja tako da u jednom trenutku može da bude aktivna samo jedna konekcija sa jednim IEC 104 master uređajem. Nakon inicijalizacije, potrebno je registrovati implementirane *handler* funkcije i pokrenuti IEC 104 slave. Dio koda zadužen za datu proceduru dat je na slici 4.19.

```

/* set the callback handler for the clock synchronization command */
CS104_Slave_setClockSyncHandler(slave, clockSyncHandler, NULL);

/* set the callback handler for the interrogation command */
CS104_Slave_setInterrogationHandler(slave, interrogationHandler, (void*) (&mb_comm_param));

/* set handler for other message types */
CS104_Slave_setASDUHandler(slave, asduHandler, (void*) (&mb_comm_param));

/* set handler to handle connection requests (optional) */
CS104_Slave_setConnectionRequestHandler(slave, connectionRequestHandler, NULL);

/* set handler to track connection events (optional) */
CS104_Slave_setConnectionEventHandler(slave, connectionEventHandler, NULL);

/* set handler for read command */
CS104_Slave_setReadHandler(slave, readHandler, (void*) (&mb_comm_param));

/* uncomment to log messages */
//CS104_Slave_setRawMessageHandler(slave, rawMessageHandler, NULL);

CS104_Slave_start(slave);

```

Slika 4.19 – Registracija handler funkcija i pokretanje IEC 104 slave uređaja

Sa slike 4.19 može se vidjeti koje su sve *callback handler* funkcije implementirane. Funkcija *clockSyncHandler* se koristi u slučaju komande za vremensku sinhronizaciju od strane IEC 104 master uređaja. Ovaj *handler* mora da ažurira trenutno vrijeme IEC 104 slave uređaja na vrijeme specificirano u sadržaju poruke. Implementacija ovog *handler*-a je data na slici 4.20.

```

static bool
clockSyncHandler (void* parameter, IMasterConnection connection, CS101_ASdu asdu, CP56Time2a newTime)
{
    printf("Process time sync command with time "); printCP56Time2a(newTime); printf("\n");

    uint64_t newSystemTimeInMs = CP56Time2a_toMsTimestamp(newTime);

    /* Set time for ACT_CON message */
    CP56Time2a_setFromMsTimestamp(newTime, Hal_getTimeInMs());

    /* update system time here */
    return true;
}

```

Slika 4.20 – Implementacija handler funkcije za sinhronizaciju vremena

Funkcija *connectionEventHandler* je opcioni *handler* koji se može koristiti u svrhe debugovanja da aplikacija informiše o događajima koji se tiču veze sa master uređajem (npr. veza otvorena ili zatvorena). Funkcija *connectionRequestHandler* je takođe opciona i takođe se može iskoristiti u svrhe debugovanja da se informiše o IP adresi master uređaja koji je poslao zahtjev za povezivanjem. Ovaj *handler* može imati i dodatnu funkcionalnost, a to je odobravanje datog zahtjeva za konekciju (npr. ako je data IP adresa označena kao dozvoljena) čime se može postići neki vid sigurnosnog mehanizma. Najvažnije *handler* funkcije su funkcije *interrogationHandler*, *asduHandler* i *readHandler*. Ove funkcije odgovaraju na konkretne zahtjeve i komande IEC 104 master uređaja i direktno manipulišu podacima preko MODBUS master API-ja. Funkcija *interrogationHandler* implementira odgovor na zahtjev za interogacijom pri čemu je moguće implementirati odgovor na interogaciju cijele stanice ili na interogaciju po grupama. S obzirom da u trenutnoj implementaciji nisu kreirane nikakve grupe, implementiran je samo odgovor na cjelokupnu interogaciju stanice.

```

static bool
interrogationHandler(void* parameter, IMasterConnection connection, CS101_ASDU asdu, uint8_t qoi)
{
    modbus_communication_param_t* mb_param = (modbus_communication_param_t*) (parameter);
    interrogation_response_t* resp = NULL;
    uint8_t idx = 0;
    uint8_t slave_idx = 0;
    uint16_t slave_id = 0;

    printf("Received interrogation for group %i\n", qoi);

    if (qoi == 20)
    { /* only handle station interrogation */

        CS101_AppLayerParameters alParams = IMasterConnection_getApplicationLayerParameters(connection);

        slave_id = (uint16_t) CS101_ASDU_getCA(asdu);
        idx = slave_id / OFFSET_BY_PORT - 1;
        if (idx < 0 || idx >= SERIAL_PORTS_NUM)
        {
            fprintf(stderr, "Invalid slave ID: %u, index out of bounds.\n", slave_id);
            IMasterConnection_sendACT_CON(connection, asdu, true);
            return true;
        }

        resp = interrogate_slave(slave_id, mb_param->slaves[idx], mb_param->num_of_slaves[idx], mb_param->ctx[idx]);
        if (resp == NULL)
        {
            fprintf(stderr, "Failed to get interrogation response for slave: %u.\n", slave_id);
            IMasterConnection_sendACT_CON(connection, asdu, true);
            return true;
        }

        IMasterConnection_sendACT_CON(connection, asdu, false);

        slave_idx = get_slave_idx(slave_id, mb_param->slaves[idx], mb_param->num_of_slaves[idx]);

        /* The CS101 specification only allows information objects without timestamp in GI responses */
        sendAllSinglePoints(connection, resp, &mb_param->slaves[idx][slave_idx]);
        sendAllScaledValues(connection, resp, &mb_param->slaves[idx][slave_idx]);

        IMasterConnection_sendACT_TERM(connection, asdu);
    }
    else
    {
        IMasterConnection_sendACT_CON(connection, asdu, true);
    }
    return true;
}

```

Slika 4.21 – Implementacija handler-a zahtjeva za interogaciju

Implementacija podrazumijeva uzimanje ASDU adrese koja odgovara adresi MODBUS slave uređaja *offset*-ovane za poziciju serijskog porta na koji je povezana, nakon čega se poziva MODBUS master funkcija za interogaciju, te se dobijene vrijednosti šalju IEC 104 master uređaju pozivom funkcija *sendAllSinglePoints* i *sendAllScaledValues*. Ovdje se jasno vidi princip mapiranja između protokola diskutovan u podglavi 4.3. Može se primjetiti da je ovdje mapiranje bilo direktno tako da se svaki diskretni ulaz i izlaz mapira kao *Single Point* IEC 104 tip, dok se vrijednosti ulaznih i izlaznih registara mapiraju pomoću *Scaled Value* tipa. Naravno, moguće je ova mapiranja proširiti u skladu sa diskusijom iz podglave 4.3, ali je za početne zahtjeve ova implementacija odgovarala. Takođe, sa slike 4.21 i 4.22 može se vidjeti i način upotrebe *iec60870* biblioteke u svrhu kreiranja ASDU poruka, dodavanja informacionih objekata i slanja istih ka IEC 104 master uređaju.


```

void sendAllSinglePoints(IMasterConnection connection, interrogation_response_t* resp, simple_slave_t* slave)
{
    CS101_ApplicationParameters alParams = IMasterConnection_getApplicationLayerParameters(connection);
    CS101_ASOU newAsdu = CS101_ASOU_create(alParams, false, CS101_COT_INTERROGATED_BY_STATION, 0, slave->id, false, false);

    for(uint8_t i = 0; i < resp->num_of_coils; i++)
    {
        InformationObject io = (InformationObject) SinglePointInformation_create(NULL, COIL_ADDRESS_START + slave->coils_addr[i],
            resp->coils[i], IEC60870_QUALITY_GOOD);
        CS101_ASOU_addInformationObject(newAsdu, io);
        InformationObject_destroy(io);
    }

    for(uint8_t i = 0; i < resp->num_of_discrete_inputs; i++)
    {
        InformationObject io = (InformationObject) SinglePointInformation_create(NULL, DISCRETE_INPUT_ADDRESS_START + slave->discrete_inputs_addr[i],
            resp->discrete_inputs[i], IEC60870_QUALITY_GOOD);
        CS101_ASOU_addInformationObject(newAsdu, io);
        InformationObject_destroy(io);
    }

    IMasterConnection_sendASDU(connection, newAsdu);
    CS101_ASOU_destroy(newAsdu);
}

void sendAllScaledValues(IMasterConnection connection, interrogation_response_t* resp, simple_slave_t* slave)
{
    CS101_ApplicationParameters alParams = IMasterConnection_getApplicationLayerParameters(connection);
    CS101_ASOU newAsdu = CS101_ASOU_create(alParams, false, CS101_COT_INTERROGATED_BY_STATION, 0, slave->id, false, false);

    // Create and send input and holding registers for testing

    for(int i = 0; i < resp->num_of_input_registers; i++)
    {
        InformationObject io = (InformationObject) MeasuredValueScaled_create(NULL, INPUT_REGISTER_ADDRESS_START + slave->input_registers_addr[i],
            resp->input_regs[i], IEC60870_QUALITY_GOOD);
        CS101_ASOU_addInformationObject(newAsdu, io);
        InformationObject_destroy(io);
    }

    for(int i = 0; i < resp->num_of_holding_registers; i++)
    {
        InformationObject io = (InformationObject) MeasuredValueScaled_create(NULL, HOLDING_REGISTER_ADDRESS_START + slave->holding_registers_addr[i],
            resp->holding_regs[i], IEC60870_QUALITY_GOOD);
        CS101_ASOU_addInformationObject(newAsdu, io);
        InformationObject_destroy(io);
    }

    IMasterConnection_sendASDU(connection, newAsdu);
    CS101_ASOU_destroy(newAsdu);
}

```

Slika 4.22 – Implementacija funkcija za slanje podataka iz interrogation handler-a

Funkcija *readHandler* služi za implementaciju odgovora na zahtjev za čitanjem IEC 104 master uređaja. Podaci za svaki slave uređaj IEC 104 masteru se prezentuju sa odgovarajućom adresom informacionog objekta za dati podatak, koja odgovara adresi datog podatka u kontekstu MODBUS adresiranja. Bitno je napomenuti da, iako se u konfiguracionom fajlu ove adrese navode od nule, konkretne adrese koje se koriste u IEC 104 komunikaciji odgovaraju adresnom modelu MODBUS protokola koji je prikazan na slici 4.7. Dakle, ako je na primjer MODBUS zahtjev da se pročita izlazni registar adrese 0 (prva adresa), IEC 104 informacioni objekat koji bi aktivirao ovaj zahtjev morao bi imati adresu koja odgovara prvoj adresi prema MODBUS modelu podataka, a to je adresa 40001. U funkciji *readHandler* ovo se koristi da bi se mogao primjeniti konkretan MODBUS master zahtjev za čitanjem, kada se otkrije kom opsegu pripada adresa informacionog objekta. Isječak koda iz *readHandler* funkcije koji ilustruje prethodnu diskusiju dat je na slici 4.23.


```

CS101_ApLayerParameters alParams = IMasterConnection_getApplicationLayerParameters(connection);
int ca = CS101_ASDU_getCA(asdu);
CS101_ASDU newAsdu = CS101_ASDU_create(alParams, false, CS101_COT_REQUEST, 0, ca, false, false);
InformationObject io = NULL;
uint8_t* state_value = NULL;
uint16_t* reg_value = NULL;
modbus_communication_param_t* mb_param = (modbus_communication_param_t*) (parameter);
uint8_t idx = ca / OFFSET_BY_PORT - 1;
if(idx < 0 || idx >= SERIAL_PORTS_NUM)
{
    fprintf(stderr, "Invalid slave ID, index out of bounds.\n");
    ioa = -1;
}
/*
IMPORTANT: Might need to further divide address space of MODBUS to fit all of the
IEC104 information elements !
*/
if(ioa >= COIL_ADDRESS_START && ioa <= COIL_ADDRESS_END)
{
    state_value = read_coil((uint16_t) ca, (uint8_t) (ioa - COIL_ADDRESS_START), mb_param->slaves[idx], mb_param->num_of_slaves[idx], mb_param->ctx[idx]);
    if(state_value == NULL)
    {
        io = NULL;
        fprintf(stderr, "Failed to read coil status, address: %i.\n", ioa);
    }
    else
    {
        io = (InformationObject) SinglePointInformation_create(NULL, ioa, *state_value, IEC60870_QUALITY_GOOD);
        printf("Reading state of the coil, address: %i\n", ioa);
    }
}
else if(ioa >= DISCRETE_INPUT_ADDRESS_START && ioa <= DISCRETE_INPUT_ADDRESS_END)
{
    state_value = read_discrete_input((uint16_t) ca, (uint8_t) (ioa - DISCRETE_INPUT_ADDRESS_START), mb_param->slaves[idx],
    mb_param->num_of_slaves[idx], mb_param->ctx[idx]);
    if(state_value == NULL)
    {
        io = NULL;
        fprintf(stderr, "Failed to read discrete input status, address: %i.\n", ioa);
    }
    else
    {
        io = (InformationObject) SinglePointInformation_create(NULL, ioa, *state_value, IEC60870_QUALITY_GOOD);
        printf("Reading state of the discrete input, address: %i\n", ioa);
    }
}
}

```

Slika 4.23 – Dio funkcije readHandler

Funkcija *asduHandler* služi za implementaciju komandi IEC 104 master uređaja za postavljanjem vrijednosti. S obzirom na iskorišćeno mapiranje (diskretni ulazi i izlazi – *Single Point*, ulazni i izlazni registri – *Scaled Value*) implementirani su odgovori samo na komande za postavljanje jedinične vrijednosti (*Single command*) i skalirane vrijednost (*Setpoint command, scaled value*). Moguće je iskoristiti komande i sa i bez vremenskog žiga (*Time tag*). U suštini, ako se primi komanda koja podešava jediničnu vrijednost, tada se pokušava sa postavljanjem vrijednosti jednog diskretnog izlaza pozivom odgovarajuće MODBUS master funkcije. Slično, komanda za postavljanje skalirane vrijednosti se prevodi u poziv MODBUS master funkcije za postavljanje vrijednosti izlaznog registra. Slično kao i kod *readHandler* funkcije, mora se voditi računa o adresiranju, odnosno primljena adresa koja odgovara MODBUS data modelu se mora provjeriti za odgovarajući opseg i translirati u adresiranje od nule da bi se koristila sa MODBUS master API funkcijama (i funkcijama same *libmodbus* biblioteke koje i vrše konkretne akcije sa podacima). Na slici 4.24 prikazan je isječak iz funkcije *asduHandler* koji daje odgovor na jediničnu komandu dobijenu od strane IEC 104 master uređaja.

```

if(CS101_ASDU_getTypeID(asdu) == C_SC_MA_1)
{
    printf("Received single command\n");

    if(CS101_ASDU_getCOT(asdu) == CS101_COT_ACTIVATION)
    {
        InformationObject io = CS101_ASDU_getElement(asdu, 0);
        if(io)
        {
            if(InformationObject_getObjectAddress(io) >= COIL_ADDRESS_START && InformationObject_getObjectAddress(io) <= COIL_ADDRESS_END)
            {
                SingleCommand sc = (SingleCommand) io;
                target_address = InformationObject_getObjectAddress(io) - COIL_ADDRESS_START;
                target_value = SingleCommand_getState(sc) == 0 ? COIL_OFF_VALUE : COIL_ON_VALUE;
                if(write_coil(slave_id, target_address, target_value, mb_param->slaves[idx], mb_param->num_of_slaves[idx], mb_param->ctx[idx]))
                {
                    printf("IOA: %i switch to %i\n", InformationObject_getObjectAddress(io), SingleCommand_getState(sc));
                    CS101_ASDU_setCOT(asdu, CS101_COT_ACTIVATION_CON);
                }
                else
                {
                    fprintf(stderr, "Failed to set coil status, address: %i.\n", InformationObject_getObjectAddress(io));
                    CS101_ASDU_setCOT(asdu, CS101_COT_UNKNOWN_IOA);
                    CS101_ASDU_setNegative(asdu, true);
                }
            }
            else
            {
                CS101_ASDU_setCOT(asdu, CS101_COT_UNKNOWN_IOA);
                CS101_ASDU_setNegative(asdu, true);
            }
            InformationObject_destroy(io);
        }
        else
        {
            printf("ERROR: message has no valid information object\n");
            return true;
        }
    }
    else
    {
        CS101_ASDU_setCOT(asdu, CS101_COT_UNKNOWN_COT);
        CS101_ASDU_setNegative(asdu, true);
    }
    IMasterConnection_sendASDU(connection, asdu);

    return true;
}

```

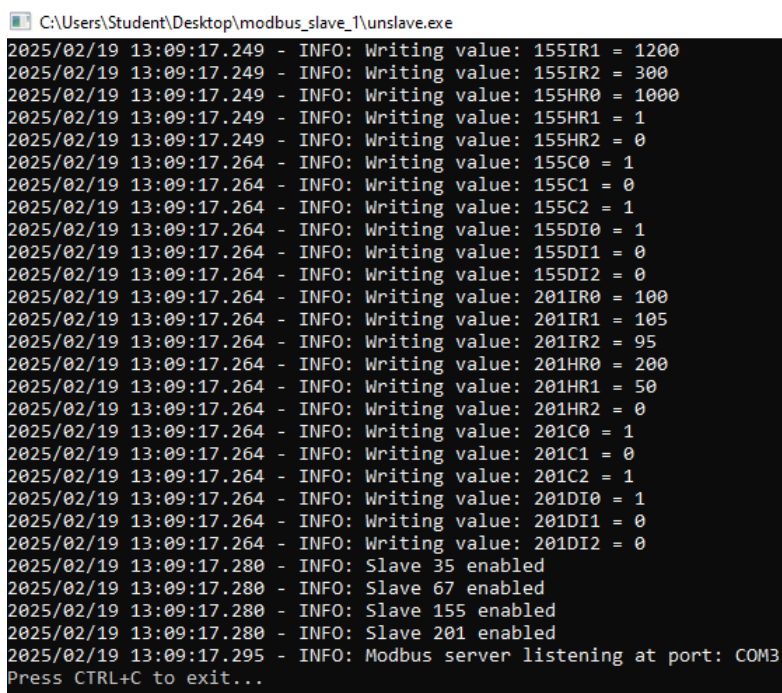
Slika 4.24 – Isječak koda iz asduHandler funkcije

Posmatrajući implementacije prethodne tri *handler* funkcije može se vidjeti da je svakoj od njih proslijeđena kao argument (prilikom registracije) instanca strukture podataka *modbus_communication_param* čiji se članovi se potom koriste za pronalaženje odgovarajućeg slave uređaja koji se adresira u zahtjevu, te za poziv funkcije iz MODBUS master API-ja za ispravno izvršenje zahtjeva nad adresiranim slave uređajem i adresiranim registrom tog slave uređaja.

Ovim je dato osnovno izlaganje implementacije aplikacije koja se izvršava na gejtveju. Sama implementacija se može dodatno proširiti, ali će mogućnosti i prijedlozi poboljšanja biti zasebno diskutovani u sklopu posljednje glave.

5. Testiranje funkcionalnosti gateway uređaja

Funkcionalnost gateway uređaja testirana je u konfiguraciji sa dva PC računara koji su povezani na ciljnu platformu na kojoj se izvršava aplikacija gejtveja. Na jednom PC uređaju aktiviran je MODBUS simulator *Unslave* [22]. Ovaj simulator je konzolni program koji omogućava kreiranje virtuelnih MODBUS slave uređaja i interakciju sa njima preko serijskih USB portova PC računara. Konfiguracija i dodavanje slave uređaja moguća je putem JSON konfiguracionog fajla (slično kao i za konfiguraciju gejtveja) i važno je da prije pokretanja testiranja konfiguracije gejtveja i ovog simulatora budu identične (identično konfigurisani svi postojeći slave uređaji). Veza između ciljne platforme i ovog PC računara je serijska, korištenjem USB-UART konvertera. Na slici 5.1 prikazana je konzola ovog simulatora nakon pokretanja sa konfiguracijom za jedan serijski port. Simulator jednostavno pročita konfiguracioni fajl, kreira virtuelne slave uređaje sa datim adresama i podešenim registrima, te popuni početne vrijednosti koje su zadate. Nakon toga se otvara serijska veza i moguće je putem odgovarajućih MODBUS poruka pristupiti i mijenjati ove podatke.



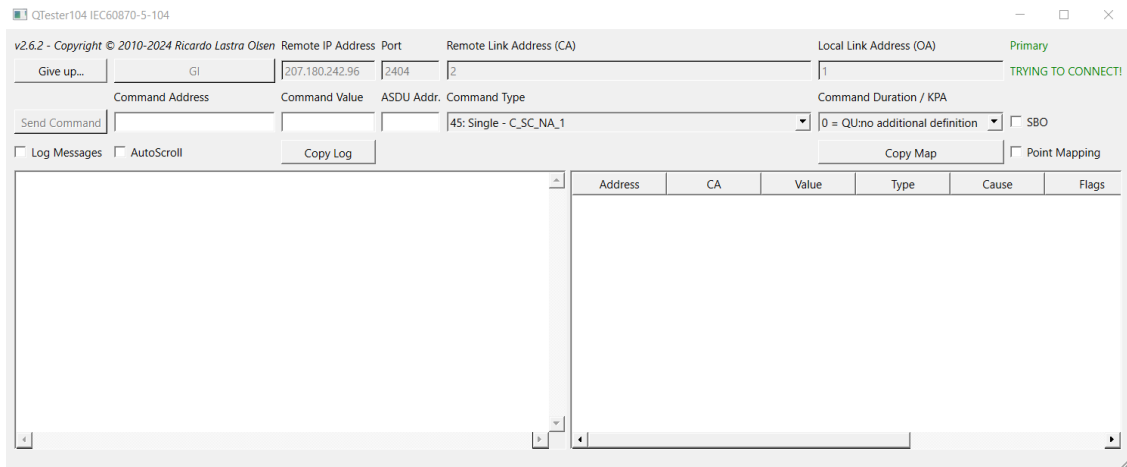
```

C:\Users\Student\Desktop\modbus_slave_1\unslave.exe
2025/02/19 13:09:17.249 - INFO: Writing value: 155IR1 = 1200
2025/02/19 13:09:17.249 - INFO: Writing value: 155IR2 = 300
2025/02/19 13:09:17.249 - INFO: Writing value: 155HR0 = 1000
2025/02/19 13:09:17.249 - INFO: Writing value: 155HR1 = 1
2025/02/19 13:09:17.249 - INFO: Writing value: 155HR2 = 0
2025/02/19 13:09:17.264 - INFO: Writing value: 155C0 = 1
2025/02/19 13:09:17.264 - INFO: Writing value: 155C1 = 0
2025/02/19 13:09:17.264 - INFO: Writing value: 155C2 = 1
2025/02/19 13:09:17.264 - INFO: Writing value: 155DI0 = 1
2025/02/19 13:09:17.264 - INFO: Writing value: 155DI1 = 0
2025/02/19 13:09:17.264 - INFO: Writing value: 155DI2 = 0
2025/02/19 13:09:17.264 - INFO: Writing value: 201IR0 = 100
2025/02/19 13:09:17.264 - INFO: Writing value: 201IR1 = 105
2025/02/19 13:09:17.264 - INFO: Writing value: 201IR2 = 95
2025/02/19 13:09:17.264 - INFO: Writing value: 201HR0 = 200
2025/02/19 13:09:17.264 - INFO: Writing value: 201HR1 = 50
2025/02/19 13:09:17.264 - INFO: Writing value: 201HR2 = 0
2025/02/19 13:09:17.264 - INFO: Writing value: 201C0 = 1
2025/02/19 13:09:17.264 - INFO: Writing value: 201C1 = 0
2025/02/19 13:09:17.264 - INFO: Writing value: 201C2 = 1
2025/02/19 13:09:17.264 - INFO: Writing value: 201DI0 = 1
2025/02/19 13:09:17.264 - INFO: Writing value: 201DI1 = 0
2025/02/19 13:09:17.264 - INFO: Writing value: 201DI2 = 0
2025/02/19 13:09:17.280 - INFO: Slave 35 enabled
2025/02/19 13:09:17.280 - INFO: Slave 67 enabled
2025/02/19 13:09:17.280 - INFO: Slave 155 enabled
2025/02/19 13:09:17.280 - INFO: Slave 201 enabled
2025/02/19 13:09:17.295 - INFO: Modbus server listening at port: COM3
Press CTRL+C to exit...

```

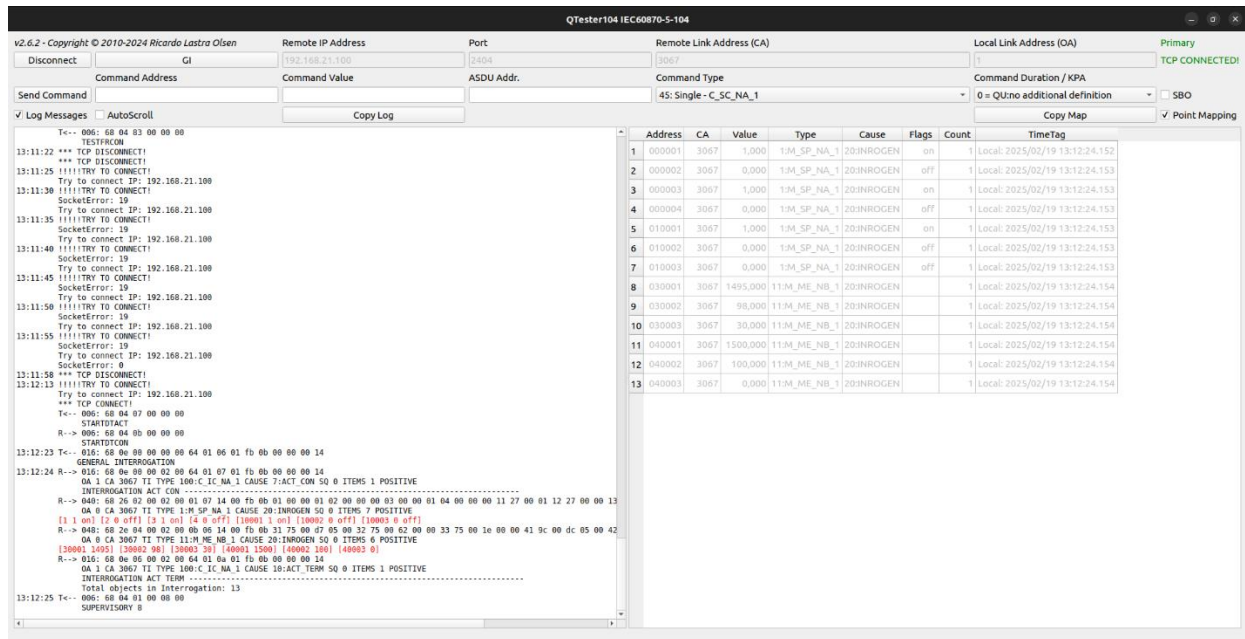
Slika 5.1 – Pokretanje MODBUS slave simulatora

Na drugom PC računaru korišten je IEC 104 Master simulator *qTester104* [23]. Ovo je IEC 104 Master simulator otvorenog koda vrlo jednostavan za korišćenje i osnovno testiranje koje je dovoljno za implementiranu aplikaciju gejtveja. Interfejs ovog simulatora nakon pokretanja prikazan je na slici 5.2.



Slika 5.2 – Interfejs qTester104 simulatora

Prvi korak jeste podešavanje IP adrese gejtvej uređaja u polje *Remote IP Address*, podešavanje porta (za IEC 104 podrazumijevano je 2404) i podešavanje adrese IEC 104 kontrolisane stanice – MODBUS slave uređaja (*Remote Link Address (CA)*). Ovaj simulator za sada podržava samo rad sa jednom IEC 104 kontrolisanom stanicom. Kada se želi poslati neka komanda, potrebno je unijeti adresu informacionog objekta u polje *Command Address*, vrijednost koja se veže za datu komandu (*Command Value*), adresu slave uređaja (*ASDU Addr.*) i iz padajućeg menija odabrati tip komande (*Command Type*). U nastavku biće prikazane osnovne komande koje su implementirane na gejtveju i odgovori na njih. Na slici 5.3 je prikaz stanja qTester104 programa nakon slanja zahtjeva za interogaciju i prijema odgovora.



Slika 5.3 – qTester104 zahtjev za interogaciju

Na slici 5.4 prikazan je ispis konzole MODBUS simulatora na kome se nalazi virtuelni slave uređaj koji je primio prethodni zahtjev.

```
2025/02/19 13:09:17.295 - INFO: Modbus server listening at port: COM3
Press CTRL+C to exit...
2025/02/19 13:12:24.517 - INFO: Modbus frame received: [67 1 0 0 1 242 232]
2025/02/19 13:12:24.533 - INFO: Reading value: 67C0 = 1
2025/02/19 13:12:24.533 - INFO: Modbus frame sent: [67 1 1 1 132 48]
2025/02/19 13:12:24.548 - INFO: Modbus frame received: [67 1 0 1 0 1 163 40]
2025/02/19 13:12:24.548 - INFO: Reading value: 67C1 = 0
2025/02/19 13:12:24.548 - INFO: Modbus frame sent: [67 1 1 0 69 240]
2025/02/19 13:12:24.564 - INFO: Modbus frame received: [67 1 0 2 0 1 83 40]
2025/02/19 13:12:24.564 - INFO: Reading value: 67C2 = 1
2025/02/19 13:12:24.564 - INFO: Modbus frame sent: [67 1 1 1 132 48]
2025/02/19 13:12:24.579 - INFO: Modbus frame received: [67 1 0 3 0 1 2 232]
2025/02/19 13:12:24.579 - INFO: Reading value: 67C3 = 0
2025/02/19 13:12:24.579 - INFO: Modbus frame sent: [67 1 1 0 69 240]
2025/02/19 13:12:24.595 - INFO: Modbus frame received: [67 2 0 0 0 1 182 232]
2025/02/19 13:12:24.595 - INFO: Reading value: 67DI0 = 1
2025/02/19 13:12:24.595 - INFO: Modbus frame sent: [67 2 1 1 116 48]
2025/02/19 13:12:24.611 - INFO: Modbus frame received: [67 2 0 1 0 1 231 40]
2025/02/19 13:12:24.611 - INFO: Reading value: 67DI1 = 0
2025/02/19 13:12:24.611 - INFO: Modbus frame sent: [67 2 1 0 181 240]
2025/02/19 13:12:24.626 - INFO: Modbus frame received: [67 2 0 2 0 1 23 40]
2025/02/19 13:12:24.626 - INFO: Reading value: 67DI2 = 0
2025/02/19 13:12:24.626 - INFO: Modbus frame sent: [67 2 1 0 181 240]
2025/02/19 13:12:24.642 - INFO: Modbus frame received: [67 4 0 0 0 1 62 232]
2025/02/19 13:12:24.642 - INFO: Reading value: 67IR0 = 1495
2025/02/19 13:12:24.642 - INFO: Modbus frame sent: [67 4 2 5 215 130 49]
2025/02/19 13:12:24.658 - INFO: Modbus frame received: [67 4 0 1 0 1 111 40]
2025/02/19 13:12:24.658 - INFO: Reading value: 67IR1 = 98
2025/02/19 13:12:24.658 - INFO: Modbus frame sent: [67 4 2 0 98 64 214]
2025/02/19 13:12:24.673 - INFO: Modbus frame received: [67 4 0 2 0 1 159 40]
2025/02/19 13:12:24.673 - INFO: Reading value: 67IR2 = 30
2025/02/19 13:12:24.673 - INFO: Modbus frame sent: [67 4 2 0 30 65 55]
2025/02/19 13:12:24.689 - INFO: Modbus frame received: [67 3 0 0 0 1 139 40]
2025/02/19 13:12:24.689 - INFO: Reading value: 67HR0 = 1500
2025/02/19 13:12:24.689 - INFO: Modbus frame sent: [67 3 2 5 220 194 130]
2025/02/19 13:12:24.704 - INFO: Modbus frame received: [67 3 0 1 0 1 218 232]
2025/02/19 13:12:24.704 - INFO: Reading value: 67HR1 = 100
2025/02/19 13:12:24.704 - INFO: Modbus frame sent: [67 3 2 0 100 193 160]
2025/02/19 13:12:24.720 - INFO: Modbus frame received: [67 3 0 2 0 1 42 232]
2025/02/19 13:12:24.736 - INFO: Reading value: 67HR2 = 0
2025/02/19 13:12:24.736 - INFO: Modbus frame sent: [67 3 2 0 0 192 75]
```

Slika 5.4 – Konzola MODBUS slave simulatora nakon odgovora na inter. zahtjev

Nakon zahtjeva za interogaciju, testirana je jedinična komanda koja postavlja vrijednost 1 („uključen“) na diskretni izlaz adrese 4 MODBUS slave uređaja. Konzola MODBUS slave simulatora nakon prijema ove komande data je na slici 5.5, a stanje IEC 104 master simulatora nakon ove komande dato je na slici 5.6.

```
2025/02/19 13:14:48.726 - INFO: Modbus frame received: [67 5 0 3 255 0 115 24]
2025/02/19 13:14:48.726 - INFO: Writing value: 67C3 = 1
2025/02/19 13:14:48.741 - INFO: Modbus frame sent: [67 5 0 3 255 0 115 24]
```

Slika 5.5 – Konzola MODBUS slave simulatora, prijem komande za setovanje coil-a

[illegible]

Slika 5.6 – Stanje *qTester104* simulatora nakon jedinične komande

S obzirom na trenutnu implementaciju gejtvej aplikacije, još je testirana komanda za postavljanje vrijednosti (*Setpoint command*). U ovom slučaju data komanda je izdata uz vremenski žig. Prikaz stanja *qTester104* simulatora nakon ove komande dat je na slici 5.7.

V2.6.2 - Copyright © 2010-2024 Ricardo Lastra Olsen
QTester104 IEC60870-5-104

Remote IP Address		Port	Remote Link Address (CA)		Local Link Address (OA)		Primary
Disconnect	GI	192.168.21.100	3067		1		TCP CONNECTED
Command Address		Command Value	ASDU Addr.		Command Type	Command Duration / KPA	
Send Command		40003	3067		62: Set-point scaled value with time tag - C_SE_TB_1	0 = Quno additional definition	SBO
✓ Log Messages		AutoScroll			Copy Map		✓ Point Mapping

```

13:23:23 T<- 016: 68 06 00 12 00 64 01 06 01 fb 00 00 00 14
GENERAL INTERROGATION
R-> 016: 68 12 00 00 00 64 01 07 01 fb 00 00 00 14
OA: 1 CA 3067 TI TYPE 100-C IC_NA 1 CAUSE 7-ACT_CON 50 0 ITEMS 1 POSITIVE
INTERROGATION ACT TERM -----
R-> 040: 68 26 14 00 00 00 01 17 14 00 fb 01 00 00 01 82 00 00 03 00 00 04 00 00 01 11 27 00 01 12 27 00 00 13
OA: 6 CA 3067 TI TYPE 1-M SP_NA 1 CAUSE 20-INROGEN 50 0 ITEMS 7 POSITIVE
[1] 1 2 0 0 01 13 0 0 01 14 1 01 10000 1 001 10002 0 0 01 10003 0 0 01
R-> 048: 68 2e 16 00 00 00 00 14 00 fb 00 31 75 00 47 05 00 32 75 00 62 00 33 75 00 1e 00 00 01 41 9c 00 dc 05 00 42
OA: 6 CA 3067 TI TYPE 11-M ME_NB 1 CAUSE 20-INROGEN 50 0 ITEMS 6 POSITIVE
[3000] 1495 [30002] 961 [30003] 70 [40001] 1500 [40002] 1000 [40003] 0
R-> 016: 68 06 20 00 00 64 01 0a 01 fb 00 00 00 14
OA: 1 CA 3067 TI TYPE 100-C IC_NA 1 CAUSE 18-ACT_TERM 50 0 ITEMS 1 POSITIVE
INTERROGATION ACT TERM -----
Total objects in Interrogation: 13
006: 68 01 00 1a 00
SUPERVISORY 1a
13:23:25 T<- 006: 68 04 43 00 00 00
TESTFRACT
R-> 006: 68 04 83 00 00 00
TESTFRCON
13:23:25 R-> 006: 68 04 43 00 00 00
TESTFRACT
T<- 006: 68 04 83 00 00 00
13:24:13 R-> 006: 68 04 43 00 00 00
TESTFRCON
T<- 006: 68 04 83 00 00 00
13:24:13 R-> 006: 68 04 43 00 00 00
TESTFRCON
T<- 006: 68 04 83 00 00 00
13:24:14 R-> 006: 68 04 43 00 00 00
TESTFRCON
T<- 006: 68 04 83 00 00 00
13:24:17 T<- 016: 68 06 00 00 1a 00 64 01 06 01 fb 00 00 00 14
GENERAL INTERROGATION
R-> 016: 68 1a 00 00 00 64 01 07 01 fb 00 00 00 14
OA: 1 CA 3067 TI TYPE 100-C IC_NA 1 CAUSE 7-ACT_CON 50 0 ITEMS 1 POSITIVE
INTERROGATION ACT TERM -----
R-> 040: 68 26 1c 00 00 00 01 17 14 00 fb 01 00 00 01 82 00 00 03 00 00 04 00 00 01 11 27 00 01 12 27 00 00 13
OA: 6 CA 3067 TI TYPE 1-M SP_NA 1 CAUSE 20-INROGEN 50 0 ITEMS 7 POSITIVE
[1] 1 2 0 0 01 13 1 01 14 1 01 10000 1 001 10002 0 0 01 10003 0 0 01
R-> 048: 68 2e 1e 00 00 00 00 14 00 fb 00 31 75 00 47 05 00 32 75 00 62 00 33 75 00 1e 00 00 01 41 9c 00 dc 05 00 42
OA: 6 CA 3067 TI TYPE 11-M ME_NB 1 CAUSE 20-INROGEN 50 0 ITEMS 6 POSITIVE
[3000] 1495 [30002] 961 [30003] 70 [40001] 1500 [40002] 1000 [40003] 0
R-> 016: 68 06 20 00 00 64 01 0a 01 fb 00 00 00 14
OA: 1 CA 3067 TI TYPE 100-C IC_NA 1 CAUSE 18-ACT_TERM 50 0 ITEMS 1 POSITIVE
INTERROGATION ACT TERM -----
Total objects in Interrogation: 13
006: 68 01 00 22 00
SUPERVISORY 22
13:24:16 T<- 006: 68 04 43 00 00 00
TESTFRCON
13:24:16 T<- 006: 68 04 43 00 00 00
TESTFRCON

```

	Address	CA	Value	Type	Cause	Flags	Count	TimeTag
1	000001	3067	1,000	1-M_SP_NA_1	20-INROGEN	on	4	Local: 2025/02/19 13:24:47.417
2	000002	3067	0,000	1-M_SP_NA_1	20-INROGEN	off	4	Local: 2025/02/19 13:24:47.418
3	000003	3067	1,000	1-M_SP_NA_1	20-INROGEN	on	4	Local: 2025/02/19 13:24:47.418
4	000004	3067	1,000	1-M_SP_NA_1	20-INROGEN	on	5	Local: 2025/02/19 13:24:47.418
5	010001	3067	1,000	1-M_SP_NA_1	20-INROGEN	on	4	Local: 2025/02/19 13:24:47.418
6	010002	3067	0,000	1-M_SP_NA_1	20-INROGEN	off	4	Local: 2025/02/19 13:24:47.418
7	010003	3067	0,000	1-M_SP_NA_1	20-INROGEN	off	4	Local: 2025/02/19 13:24:47.418
8	030001	3067	1495,000	11-M_ME_NB_1	20-INROGEN	off	4	Local: 2025/02/19 13:24:47.420
9	030002	3067	98,000	11-M_ME_NB_1	20-INROGEN	off	4	Local: 2025/02/19 13:24:47.420
10	030003	3067	30,000	11-M_ME_NB_1	20-INROGEN	off	4	Local: 2025/02/19 13:24:47.420
11	040001	3067	1500,000	11-M_ME_NB_1	20-INROGEN	off	4	Local: 2025/02/19 13:24:47.420
12	040002	3067	100,000	11-M_ME_NB_1	20-INROGEN	off	4	Local: 2025/02/19 13:24:47.420
13	040003	3067	255,000		62	7 pos exe	5	Field: 25/02/19 13:25:26.000 ok

Slika 5.7 – Stanje *qTester104* simulatora nakon komande za postavljanje vrijednosti

Izvršavanje prethodne komande na MODBUS slave simulatoru prikazano je na slici 5.8.


```
2025/02/19 13:25:26.464 - INFO: Modbus frame received: [67 6 0 2 0 255 103 104]  
2025/02/19 13:25:26.464 - INFO: Writing value: 67HR2 = 255  
2025/02/19 13:25:26.464 - INFO: Modbus frame sent: [67 6 0 2 0 255 103 104]
```

Slika 5.8 – Konzola MODBUS simulatora, prijem komande za postavljanje vrijednosti izlaznog registra

Ovim je prikazano testiranje svih osnovnih komandi koje su implementirane trenutno u sklopu gejtvej aplikacije. Izostao je jedino odgovor na zahtjev za čitanje iz razloga što simulator *qTester104* nije implementirao datu funkcionalnost (sam zahtjev za čitanje u IEC 104 protokolu nije čest i, štaviše, često se izbjegava). Dakle, nakon datog testiranja može se utvrditi da gejtvej ispravno obavlja implementirane funkcionalnosti.

6. Zaključak i predlozi poboljšanja

Gateway uređaj između protokola IEC 104 i MODBUS je vrlo koristan i neophodan uređaj u svim industrijskim i elektroenergetskim sistemima koji imaju potrebu za integracijom MODBUS uređaja i podsistema sa SCADA sistemima za nadzor i kontrolu. S obzirom na različitosti protokola ali i njihove popularnosti i velikog broja uređaja koji ih implementira, potreba za ovakvim gejtvej uređajima je sasvim opravdana. Trenutna implementacija gejtveja omogućava jednostavnu upotrebu na osnovnom nivou sa najjednostavnije izvršenim mapiranjem između dva protokola. U nekim slučajevima, ovakva implementacija može biti nedovoljna, odnosno potrebno je izvršiti kompleksniju implementaciju i proširiti ju na više mogućnosti implementacije mapiranja. Dakle, prvi korak ka poboljšanju rada gejtveja jeste modifikacija ili dodavanje opcija u konfiguracionim fajlovima, tako da se može dati više informacija o nekom registru, diskretnom ulazu ili izlazu da bi se moglo izvršiti preciznije mapiranje između tipova podataka ova dva protokola. Trenutno se svaki diskretni ulaz i izlaz MODBUS protokola mapira u jediničnu tačku (*Single point*) IEC 104 protokola. Međutim, moguće je kroz odgovarajuću konfiguraciju podesiti da se grupišu dva diskretna ulaza ili izlaza tako da se može uvrstiti i IEC 104 objekat tipa dvostruke tačke (*Double point*). Slično, mogli bi se konfigurisati i parovi ulaznih ili izlaznih registara koji bi onda mogli prezentovati podatak realnog broja jednostruke preciznosti ili stanje binarnog brojača koji su definisani u IEC 104 protokolu. Ova mapiranja bi dovela do proširenja implementacije sa strane parsiranja konfiguracionog fajla, implementacije pojedinih struktura podataka i implementacije dodatnih funkcionalnosti odgovora na zahtjeve koji sada mogu nastati za novododane tipove podataka. Naravno, sve ovo zavisi od konkretnih fizičkih MODBUS slave uređaja koji se povezuju u sistem. Ako oni sami imaju registre u koje bi upisivali ovakve vrijednosti po svojoj specifikaciji, onda ima smisla praviti dodatna mapiranja. Dakle, većina predloga poboljšanje tiče se proširenja same implementacije tako da se podrži mogućnost detaljnije konfiguracije, više tipova podataka definisanih IEC 104 protokolom i više odgovara na komande i zahtjeve IEC 104 master uređaja. Moguće je takođe dodati i konfiguracije za određene grupe podataka, tako da se mogu kreirati odgovori na zahtjeve grupnih interogacija kao i ostali detalji koji postoje u IEC 104 protokolu, a nemaju direktnu relaciju sa nekim konceptom iz MODBUS protokola. Konačno, testiranje na stvarnim fizičkim sistemom i fizičkim uređajima bi bilo neophodno za konačno potvrđivanje funkcionalnosti kao i dokaz mogućnosti da sistem odgovori na postavljene zahtjeve u realnom vremenu i okruženju.

Za kraj je korisno još dati napomenu da su, pored ovog izvještaja, sva implementacija, dokumentacija i načini upotrebe definisani na GitHub stranici projekta koja će takođe da sadrži sva buduća ažuriranja na temu ovog projekta [20].

Literatura

- [1] <https://gateway-iot.com/modbus-to-iec104-p00426p1.html>, posjećeno: 20. 2. 2025.
- [2] <https://www.nuvoton.com/products/microprocessors/arm9-mpus/nuc980-industrial-control-iot-series/>, posjećeno: 20. 2. 2025.
- [3] NUC980 Series Technical Reference Manual, *Nuvoton*, Jan. 07, 2020.
- [4] MAX13487E/MAX13488E: Half-Duplex RS-485-Compatible Transceiver with AutoDirection Control Data Sheet, *Analog Devices*, Rev 3., Aug. 2024.
- [5] CA-IS372x High-Speed Dual-Channel Digital Isolators, *Chipanalog Microelectronics*, Rev 1.08, Dec. 17, 2024.
- [6] https://github.com/OpenNuvoton/MA35D1_Buildroot, posjećeno: 22. 2. 2025.
- [7] <https://buildroot.org/>, posjećeno: 22. 2. 2025.
- [8] NUC980 NuWriter User Manual, *Nuvoton*, Apr. 22, 2019.
- [9] <https://www.u-boot.org/>, posjećeno 22. 2. 2025.
- [10] Linux Device Drivers, *Jonathan Corbet, Alessandro Rubini, Greg Kroah-Hartman*, O'Reilly Media Inc. February 2005.
- [11] NUC980 Linux 5.10 BSP User Manual, *Nuvoton*, Sep. 1, 2023.
- [12] <https://docs.kernel.org/devicetree/usage-model.html>, posjećeno: 22. 2. 2025.
- [13] <https://cdn.kernel.org/pub/linux/kernel/projects/rt/5.10/>, posjećeno: 22. 2. 2025.
- [14] Description and analysis of IEC 104 Protocol, Technical Report, *Petr Matoušek*, Faculty of Information Technology, Brno University of Technology, Brno, Czech Republic, December 2017.
- [15] <https://github.com/mz-automation/lib60870>, posjećeno: 24. 2. 2025.
- [16] MODBUS Application Protocol Specification V1.1b3, *The Modbus Organization*, Apr. 26, 2012.
- [17] <https://medium.com/nerd-for-tech/a-brief-walkthrough-of-modbus-with-diagrams-a0bd4133f370>, posjećeno: 24. 2. 2025.
- [18] <https://libmodbus.org/>, posjećeno: 24. 2. 2025.
- [19] <https://www.json.org/json-en.html>, posjećeno: 25. 2. 2025.
- [20] https://github.com/lukavidic/iec104_modbus_gateway, posjećeno 25. 2. 2025.
- [21] <https://jansson.readthedocs.io/en/latest/>, posjećeno: 25. 2. 2025.

[22] <https://unserver.xyz/docs/unslave/>, posjećeno: 26. 2. 2025.

[23] <https://github.com/riclolsen/qttester104>, posjećeno 26. 2. 2025.