Sustav za podešenje i nadzor parametara digitalnog sustava upravljanja s procesorom AM2634

Modrić, Jura

SVEUČILIŠTE U RIJECI

TEHNIČKI FAKULTET

Diplomski sveučilišni studij elektrotehnike

Diplomski rad

UPRAVLJANJE ASINKRONIM STROJEM POMOĆU TI AM2634 PLATFORME

Rijeka, siječanj 2024.

Jura Modrić
0069055073

SVEUČILIŠTE U RIJECI

TEHNIČKI FAKULTET

Diplomski sveučilišni studij elektrotehnike

Diplomski rad

UPRAVLJANJE ASINKRONIM STROJEM POMOĆU TI AM2634 PLATFORME

Mentor: Prof. dr. sc. Neven Bulić

Komentor: Dr. sc. Dominik Cikač

Rijeka, siječanj 2024.

Jura Modrić
0069055073

IZJAVA

Sukladno članku 8. Pravilnika o diplomskom radu, diplomskom ispitu i završetku diplomskih sveučilišnih studija Tehničkog fakulteta Sveučilišta u Rijeci od 1. veljače 2020. godine, izjavljujem da sam samostalno izradio diplomski rad.

ZAHVALE

Ovo istraživanje bilo je putovanje ispunjeno izazovima i postignućima. Zahvaljujem mentoru na pruženoj prilici za rad na zanimljivome projektu. Zahvaljujem i kompaniji Danieli Systec te svima koji su na direktan ili indirektan način bili uključeni u cijeli proces. Bez podrške i suradnje svih navedenih projekt ne bi mogao biti realiziran.

SADRŽAJ

1. UVOD

Rad istražuje načine upravljanja mikrokontrolerima s naglaskom na uspostavljanje komunikacije i razmjene podataka s ugradbenim sustavima od strane korisnika na jednostavan način kroz grafičko sučelje. Nakon upoznavanja s razvojnim okruženjem i mogućnostima ove platforme, započet je razvoj sustava s naglaskom na funkcionalnost kontrole električnoga stroja i prikupljanje podataka te komunikacije preko mreže. Cilj je razviti sustav koji omogućava komunikaciju u stvarnom vremenu između korisnika i mikrokontrolera te upravljanog električnog stroja preko *TCP/IP/Ethernet* [1] [2] [3] protokola za komunikaciju. Sustav omogućuje prijenos i obradu podataka te upravljanje asinkronim motorom preko digitalne upravljačke jedinice.

Mikrokontroleri služe kao procesne jedinice ugradbenih sustava. Korištenjem biblioteka za upravljanje perifernih jedinica te povezivanjem s ostalim vanjskim modulima implementirane su razne funkcionalnosti koje procesnim jedinicama omogućuju interakciju s fizičkim svijetom. Mikrokontroleri su dizajnirani za učinkovito izvršavanje specifičnih zadataka u stvarnome vremenu, što ih čini idealnim za aplikacije koje zahtijevaju preciznu kontrolu i brzi odziv.

Promatrani mikrokontroler baziran je na višejezgrenoj ARM arhitekturi. Uključuje značajke kao što su višejezgrena obrada, upravljanje u stvarnome vremenu, mogućnosti povezivanja preko komunikacijskih kanala i različitih protokola s drugim digitalnim sustavima i perifernim uređajima. Posjeduje dva MB radne memorije, raspodijeljene na četiri dijela po 512 KB, kako bi svaka jezgra imala vlastitu dediciranu memoriju. Jezgre mogu međusobno komunicirati upotrebom međuprocesorske komunikacije (IPC – *inter procesor communication*). Kako bi se to ostvarilo, potrebna je konfiguracija adresnoga prostora radne memorije, to jest zajedničkoga dijela u kojemu jezgre mogu razmjenjivati podatke.

Provedeno je i istraživanje integracije relacijskih te *in-memory* baza podataka za pohranjivanje i dohvaćanje podataka za analizu prikupljenih s motora.

1.1. Korištene tehnologije

Kao primarni alati u razvoju korišteni su *Docker* [4], *Git* [5], *C# (Visual Studio)* [6], *C (Code Composer Studio)* [7] i *SQL* [8]. *Docker* je platforma za podjelu aplikacije na dijelove, module ili kontejnere. Kontejneri su izolirana okruženja koja uključuju sve što je potrebno za izvođenje aplikacije (kod, izvođače, biblioteke i ostale alate sustava).

Docker pruža način za implementaciju aplikacija u različitim okruženjima, od razvoja do produkcije. *Visual Studio* nudi integraciju s Dockerom, olakšavajući kompilaciju koda te ispravljanje pogrešaka. Koraci za integraciju *Dockera* podrazumijevaju konfiguraciju potrebnih parametara kako bi *Docker* znao kreirati potrebno okruženje. Okruženje može biti *Linux* ili *Windows*. *Docker Desktop* kao samostalna aplikacija pruža korisničko sučelje za upravljanje na računalima s *Windows* i *macOS* operacijskim sustavima. Emulator se pokreće paralelno s Visual Studio okruženjem. *Dockerfile* [9] je konfiguracijska datoteka te postoji za svaku komponentu (bazu podataka, uslugu, korisničko sučelje) koja bi trebala biti u kontejneru. *Dockerfile* datoteka sadrži upute za inicijalizaciju virtualne mašine u kojoj će se izvoditi aplikacijski kod. Uključuje kreiranje *image* datoteke koja predstavlja virtualno računalo i u kojoj su pohranjeni podaci vezani za instancu virtualnoga računala, njegove neophodne biblioteke i postavke. Poanta ovoga pristupa je postizanje neovisnog izvođenja programa s obzirom na strojnu opremu na kojoj se program izvodi. Primjer *Dockerfile* datoteke za *.NET Core* [10] servis:

```
FROM mcr.microsoft.com/dotnet/aspnet:5.0 AS base
WORKDIR /app
EXPOSE 80
FROM mcr.microsoft.com/dotnet/sdk:5.0 AS build
WORKDIR /src
COPY ["Service/Service.csproj", "Service/"]
RUN dotnet restore "Service/Service.csproj"
COPY . .
WORKDIR "/src/Service"
RUN dotnet build "Service.csproj" -c Release -o /app/build
FROM build AS publish
RUN dotnet publish "Service.csproj" -c Release -o /app/publish
FROM base AS final
WORKDIR /app
COPY --from=publish /app/publish .
ENTRYPOINT ["dotnet", "Service.dll"]
```

Primjer datoteke *docker-compose.yml* kojom definiramo kontejnere u *Dockeru*

```
version: '3.8'
services:
  database:
    image: database-image
    ports:
      - "5432:5432"
  service:
    build:
      context: ./Service
    ports:
      - "5000:80"
    depends_on:
      - database
  gui:
    build:
      context: ./GUI
    ports:
      - "8080:80"
    depends_on:
      - service
```

Naredbe za rad s *Docker Compose* dane su u nastavku. Treba napomenuti da su ovo samo osnovne naredbe, a uz samu naredbu mogu se koristiti i dodatni parametri:

- *docker-compose up*: pokreće kontejnere definirane u datoteci *docker-compose.yml*
- *docker-compose down*: zaustavlja i uklanja kontejnere definirane u datoteci docker-compose.yml
- *docker-compose build*: izgrađuje ili ponovno izgrađuje izvršni kod u slučaju promjene u *Dockerfile* datoteci.

Ostali parametri vezani uz ove naredbe, pa i druge naredbe, mogu se pronaći pomoću *help* naredbe u *Windows Terminal* konzolnoj aplikaciji.

Bitbucket [11] omogućuje praćenje svih promjena u kodu. Uzimajući u obzir široki aspekt uključenoga koda i prirodu ovoga projekta, praćenje svih promjena na jednome mjestu uvelike olakšava razvoj projekta, rješavanje problema te održavanje integriteta koda. Upravljanje verzijama koda pomaže u rješavanju problema i održavanju stabilnosti projekta. Mogu se i vratiti prethodne verzije koda ako se pojavi problem. To olakšava praćenje napretka u rješavanju problema i osigurava da se promjene u kodu odnose na konkretne zadatke. Ukoliko je napravljena izmjena na pojedinome dijelu koda te se zbog pojave pogreške ne može pokrenuti, jednostavno je vratiti prijašnju verziju i otkloniti problem oko pokretanja novih promjena.

Kontrolom unutar *Visual Studio* razvojnog okruženja postiže se integracija s alatima za upravljanje verzijama kao što je *Git*, ali može raditi i s *BitBucket* platformom. Verzijama se može upravljati izravno pomoću *Visual Studio* alata koristeći naredbe poput *commit, push* i *pull*.

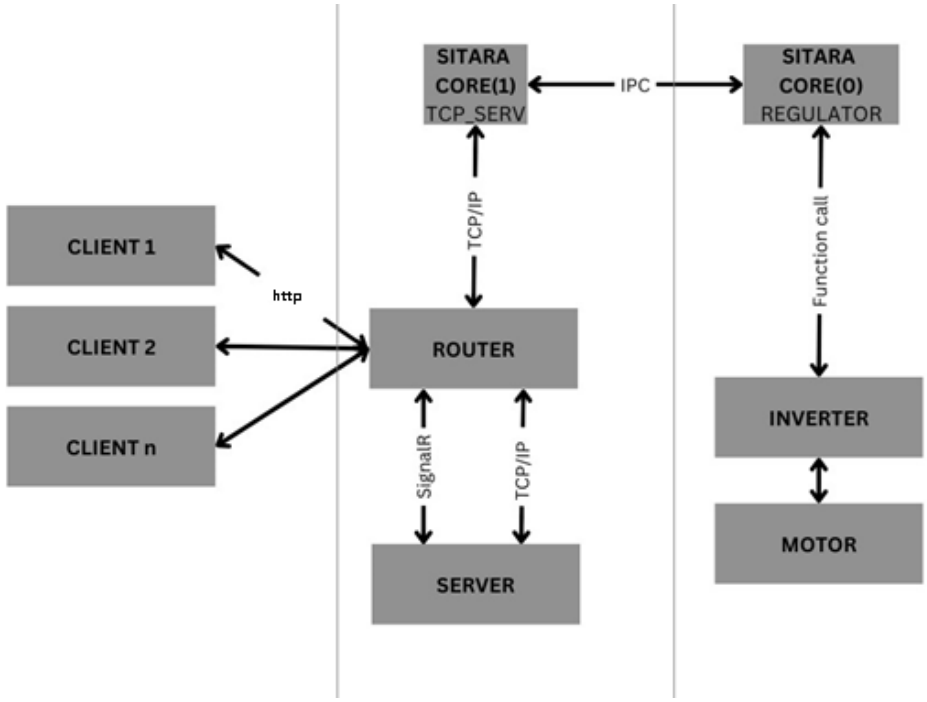
Visual Studio i *Visual Studio Code* korišteni su za razvoj poslužiteljskih i korisničkih dijelova sustava. Visual Studio pruža alate za razvoj *C#* aplikacija, dok *Visual Studio Code* omogućava razvoj sučelja koristeći *HTML*, *CSS* i *JavaScript* jezicima. Kako bi se olakšao pristup i upravljanje podacima u *SQL* bazi, koristili smo *Entity Framework* Core. Ovaj *ORM* (*Object-Relational Mapping*) alat pojednostavljuje rad s bazom podataka kroz *.NET* aplikacije.

Za razvoj mikrokontrolerskoga dijela sustava korišten je *Code Composer Studio* [7] ili skraćeno *CCS*. Pruža podršku za razvoj ugradbenih sustava i mikrokontrolera, namijenjen je za sve procesore i mikrokontrolere tvrtke Texas Instruments. U projektu je korišten C jezik za programiranje mikrokontrolera. *Code Composer Studio* omogućava razvoj logike i algoritama potrebnih pri kontroli motora i prikupljanju podataka s njega. *Postman* [12] i *Swagger* [13] alati korišteni su za provjeru uspješnoga slanja podataka između dijelova podsustava.

Inicijalno su zahtjevi ručno slani servisu pomoću *GET* i *POST HTTP* metoda. Kasnije je taj dio sustava nadograđen *SignalR Hub*-om [14] koji koristi *WebSocket* [15] tehnologiju za brz i asinkroni prijenos podataka tako da ostali procesi unutar sustava ne moraju čekati dolazak informacije i time blokirati izvođenje. *Postman* i *Swagger* su samo alati za praktičnije i konciznije ispitivanje zahtjeva naspram upisivanja pristupne adrese (u traku za adrese u pregledniku) te ispisivanja podataka u *raw* formatu. Ovi alati tako imaju podršku za *JSON* [16] format te olakšavaju razvoj jasnim prikazom i pregledom podataka koje šaljemo ili primamo. Tijekom razvoja razmišljalo se o konceptu sustava u stvarnome vremenu. Sustavi u stvarnome vremenu računalni su sustavi dizajnirani za odgovor na vanjske događaje unutar unaprijed definiranoga vremenskog okvira. Sustave u stvarnome vremenu karakterizira njihova sposobnost pružanja deterministčkih i vremenski predvidivih odgovora u programskim rješenjima u kojima su vremenska ograničenja kritična. Sustavi u stvarnome vremenu važni su i za osiguravanje precizne i pravovremene kontrole povezanih uređaja kao što su motori i senzori s minimalnim, konstantnim i predvidivim vremenom kašnjenja.

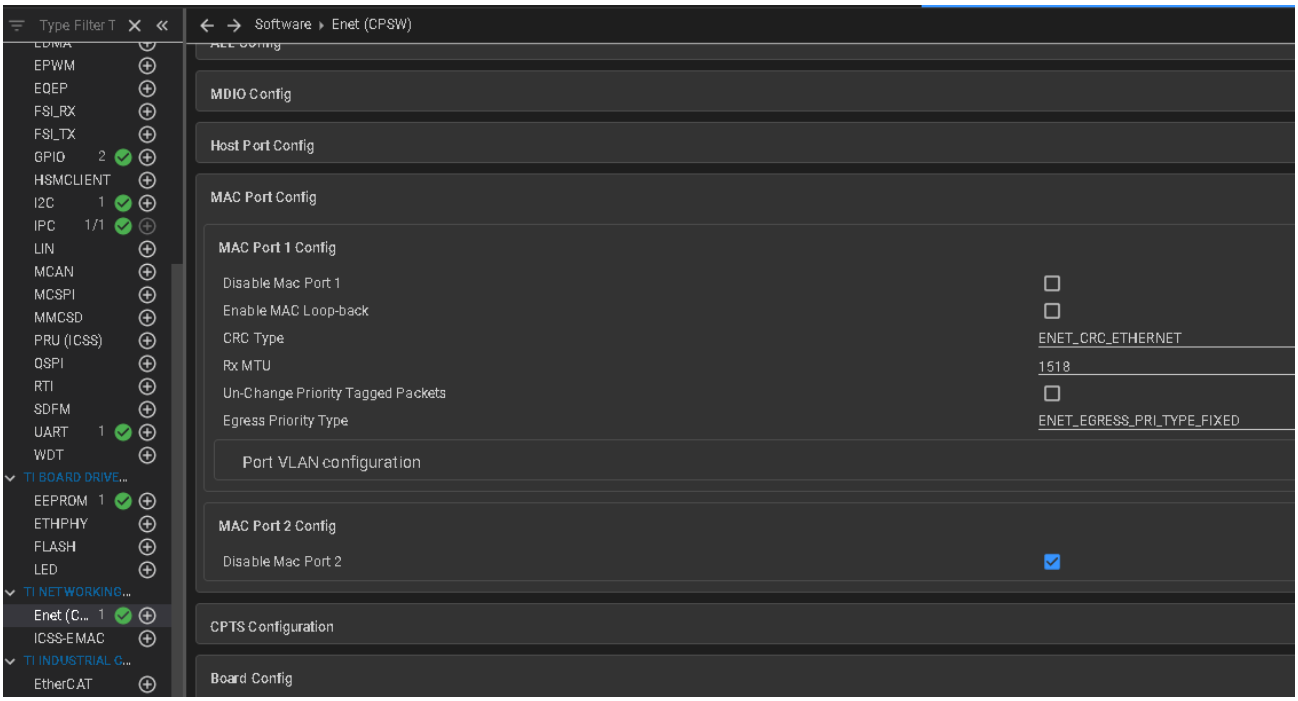
2. ARHITEKTURA SUSTAVA I MREŽE

Arhitektura sustava i mreže predstavlja okvir za međusobno povezivanje hardverskih i softverskih komponenti. Implementirane su softverske komponente (moduli aplikacije), uključujući module za upravljanje motorima te komunikaciju s ostalim modulima poput mikrokontrolera. Arhitektura sustava temelji se na strukturiranom pristupu te sadrži mehanizme za interprocesorsku komunikaciju u stvarnome vremenu. Omogućava precizno upravljanje pretvaračem na koji je spojen električni stroj uz prilagodbu brzine, smjera rotacije te ostalih parametara poput iznosa struje u pojedinoj fazi stroja. Na Slici 1. prikazan je grafički prikaz entiteta sustava te njihove međusobne veze, uz naziv korištenoga komunikacijskog protokola.



Slika 1. Funkcionalna arhitektura sustava

Konfiguracija potrebnih modula vrši se preko ugrađenog alata unutar *CSS* aplikacije. Taj alat zove se *Sysconfig* [17] i svrha mu je olakšavanje inicijalizacije potrebnih modula na razvojnoj pločici. Postiže se modularnost funkcija mikrokontrolera na način da možemo uključivati pojedine funkcije mikrokontrolera bez ovisnosti od ostalih nepotrebnih funkcija. *Sysconfig* alat objedinjuje sve parametre potrebne za inicijalizaciju pojedine funkcionalnosti pločice na jedno mjesto kroz grafičko sučelje. Na Slici 2. prikazana je konfiguracija *ethernet* modula kako bi se stekao dojam o načinu inicijalizacije modula.



Slika 2. Sysconfig konfiguracijski alat unutar CCS-a

U izborniku s lijeve strane nalazi se popis dostupnih modula i funkcija koje razvojna pločica podržava. Novi moduli se dodaju pritiskom na simbol plus, a njihova prisutnost simbolizirana je zelenom kvačicom te brojem ukoliko je moguće postaviti više modula istoga tipa. U glavnome se prozoru nalaze svi dostupni parametri za odabrani modul. Za *ethernet* modul možemo vidjeti mogućnost konfiguracije *MAC* (*Media Access Control*) [18] portova za komunikaciju sa sljedećim slojem u mrežnome modelu. Na isti način konfigurirani su i ostali parametri i moduli koji se izvršavaju na mikrokontroleru.

Komunikacijski protokoli definiraju pravila i konvencije za razmjenu podataka između entiteta unutar sustava. Razumijevanje mrežnih protokola neophodno je za izradu sustava. *TCP/IP* ili *Transmission Control Protocol* te *Internet Protocol* pružaju mogućnost pouzdane komunikacije preko mreže i s pripadajućim metodama služe kao osnova za uspostavljanje pouzdanih mrežnih kanala između aplikacije i mikrokontrolera.

TCP, IP i Ethernet su različiti protokoli s različitim zadaćama, ali su usko povezani u smislu neophodnosti svakoga od njih kada je u pitanju prijenos podataka u računalnim mrežama. Model mreže sustava podijeljen je na slojeve prikazane u Tablici 1. Svaki promatrani protokol nalazi se u drukčijem sloju te je odgovoran za implementaciju pojedinih potrebnih funkcionalnosti.

Tablica 1. OSI model mreže

Aplikacijski sloj (App)
Prezentacijski sloj (NDR)
Sloj sesije (DNS)
Transportni sloj (TCP)
Mrežni sloj (IP)
Sloj podatka (MAC)
Fizički sloj (Ethernet)

Ethernet je tehnologija koja se odnosi na fizički sloj mrežnoga modela. Omogućuje komunikaciju između različitih uređaja unutar mreže, uključujući komunikaciju s mikrokontrolerom i drugim uređajima u okolini. Svi uređaji spojeni su na usmjerivač ili ruter. Pri tome je mikrokontroler spojen pomoću e*thernet* kabela na prvi *ethernet* port na usmjerivaču, dok je računalo s razvojnim okruženjem spojeno na drugi port usmjerivača. Razvojna pločica posjeduje dva *ethernet* porta konfigurirana kako je prethodno pokazano pomoću *SysConfig* alata. *Ethernet* se koristi za povezivanje računalnih uređaja (računala, serveri, usmjerivači, prekidači itd.) u istoj lokalnoj mreži ili kraće *LAN-u*. Uključuje komponente poput mrežnih kabela i sklopki (*switch*) kako bi omogućio komunikaciju između uređaja na istoj mreži, zajedno s *MAC* adresama za identifikaciju uređaja na lokalnoj mreži.

TCP i *UDP* su protokoli u transportnom sloju mrežnoga modela. *TCP* je protokol odgovoran za pouzdan prijenos podataka između računala putem *IP* adresa, uz pomoć istoimenoga protokola. Omogućuje uspostavu i upravljanje vezama između računala, segmentaciju podataka na manje dijelove, numeriranje tih segmenata radi ispravnoga sastavljanja na odredištu, kontrolu toka kako bi se izbjegla zagušenja u mreži te pouzdano potvrđivanje dostave podataka. Za pravilan rad potrebno je namjestiti postavke *DHCP* [19] poslužitelja koji dodjeljuje adrese uređajima spojenim

na mrežu. Potrebno je pravilno konfigurirati *IP* adrese i portove na kojima će biti ostvarena komunikaciju između pojedinih slojeva aplikacije. *TCP* se koristi na višem, transportnom sloju kako bi omogućio pouzdanu komunikaciju između mikrokontrolera i drugih uređaja na mreži. Ovaj protokol osigurava da se podatci ispravno segmentiraju, šalju i ponovno organiziraju u smislenu cjelinu na odredištu. Nakon slanja i zaprimanja svaka strana u komunikaciji je odgovorna za potvrdu istih. *Ethernet, TCP i IP* protokoli surađuju kako bi omogućili komunikaciju između uređaja na *LAN* mreži uz pouzdan i siguran prijenos podataka.

SignalR je biblioteka koja objedinjuje metode, funkcije i sučelja kako bi omogućila interakciju u stvarnome vremenu. Pojednostavljuje upotrebu dvosmjerne komunikacije između mrežnih aplikacija i klijenata u stvarnome vremenu. Posebno je pogodan za scenarije u kojima je bitna asinkrona komunikacija vrlo maloga, ali neizbježnoga vremena kašnjenja. *SignalR* koristi *evente* ili događaje. To znači da se izvršavanje metoda unutar algoritma ne događa bazirano na vremenskom intervalu, već kada se definirani događaj ostvari. Međusloj zadužen za obradu događaja generira te obavještava sve pretplatnike (*engl. subscribers*) o tom događaju. Tada se izvršava metoda za ostvarivanje zahtjeva klijenta u stvarnome vremenu.

Kako svaki sloj i svaka komponenta unosi neko realno, ali vrlo malo vrijeme kašnjenja te kako komponente sustava dolaze od različitih proizvođača, za posljedicu imamo činjenicu da i svaki sloj ima neku minimalnu vrijednost za koju možemo reći da se takvo vrijeme izvođenja smatra stvarnim vremenom. Tako u kontekstu *SignalR* protokola smatramo da je sustav izvođenja u stvarnome vremenu takav da poslužitelj usluge (u našem slučaju servis kao računalni program koji prikuplja podatke s mikrokontrolera) distribuira prema klijentu podatke istoga trenutka (u što kraće mogućem definiranom vremenu) kada te podatke zaprimi od treće strane.

SignalR koristi *WebSocket* protokol za rad. *WebSocket* je protokol koji koristi *TCP* protokol za ostvarivanje dvosmjerne komunikacije između klijenta i poslužitelja. Oni uspostavljaju neprekidnu vezu eliminirajući potrebu za ponovnim uspostavljanjem veze nakon svake iteracije slanja paketa. To dovodi do ukupno bržega prijenosa podataka. *WebSocket* radi na istim portovima kao i *HTTP* protokol, čime su izbjegnute komplikacije oko konfiguracija dodatnih portova te brige o sigurnosti za iste. Koristi *full-duplex* način rada, što znači da je podržano paralelno primanje i

slanje podataka, za razliku od *HTTP* protokola koji istovremeno može samo ili poslati ili zaprimiti podatkovni paket.

U *Hub.cs*, kako se naziva glavna klasa, nalaze se svi potrebni modeli i metode za izvršavanje komunikacije prema više uređaja. Tako se na poslužitelj, koji pruža komunikaciju od mikrokontrolera prema ostatku mreže, može spojiti više korisnika. Korisnička aplikacija može se pokrenuti s više uređaja te se istovremeno spojiti na poslužitelj, a zahvaljujući SignalR-u komunikacija prema svima bit će u stvarnome vremenu, ovisno o mrežnim sposobnostima poslužitelja. Taj dio sustava pruža komunikacijski kanal između mikrokontrolera i ostalih uređaja kako bi se rasteretio mikrokontroler od komunikacije jedan na prema više korisnika.

Osnovna namjena mikrokontrolera je upravljanje elektroničkim energetskim pretvaračem na koji je spojen električni stroj. Interakcija s mikrokontrolerom se svodi na slanje i primanje naredbi i podataka putem računalnoga sučelja. Međuslojevi zaduženi za upravljanje komunikacijom na poslužitelju brinu o istovremenom pristupu većega broja korisnika. Ovaj pristup omogućava oslobađanje resursa na mikrokontroleru i povećava razinu sigurnosti aplikacije za rad u stvarnome vremenu izolirajući je u zasebnoj jezgri. Korisnici u ovome kontekstu ne moraju nužno biti osobe, to mogu biti i vanjski sustavi koji uzimaju podatke s poslužitelja i koriste ih u daljnje svrhe; za obradu podataka, spremanje ili korištenje za proračun vrijednosti u ostalim sustavima koji bi se eventualno mogli spojiti na ovaj sustav upravljanja motorom i sl. Tako u primjeni pogona za dizalo sustav upravljanja motorom podiže teret i kao korisnik ima sustav višega reda koji promatra na kojoj visini se nalazi teret, koliko je snage utrošeno u podizanje toga tereta itd.

SQL baza podataka koristi se za pohranu i dohvaćanje informacija povezanih s kontrolom i nadzorom mikrokontrolera. Oslanjajući se na principe upravljanja bazama podataka koristimo *SQL* upite kako bismo integrirali operacije nad bazama podataka. To omogućuje izvođenje operacija za pohranu podataka koje se neizravno pozivaju od samoga servisa. Na poslužitelju je odgovornost dohvaćanja i obrade podataka s mikrokontrolera dok je zadaća mikrokontrolera samo odgovarati na njih. Implementirane su funkcije za komunikaciju s *Redis* bazom (samo u svrhu

testiranja funkcionalnosti). To je *in-memory* baza podataka, naspram *SQL* baze koja je relacijska baza. Takve baze zovemo još i *NoSQL* [20].

2.1. Poslužitelj i servisi

Kod koji se izvodi na poslužitelju - serveru omogućava komunikaciju između korisničkoga sučelja i servisa, poput komunikacije s upravljačkim modulom motora, obradu podataka i pružanje informacija korisnicima. Kada poslužitelj nije dostupan na mreži, modularni dizajn omogućava da ostale komponente rade nesmetano i neovisno jedna o drugoj. Korisnici imaju interakciju sa serverom putem mrežnoga preglednika i šalju zahtjeve za promjenom parametara motora ili za prikazom informacija o motoru.

Poslužitelj zaprima zahtjev te izvršava pripadajuću metodu kako bi zaprimljeni zahtjev obradio. Nakon obrade generirani odgovor šalje se natrag prema korisničkom sučelju. Odgovori uključuju podatke o statusu, informacije o motoru i ostalim dijelovima sustava. Poslužitelj je odgovoran za povezivanje s bazom podataka kako bi spremio i dohvaćao informacije o motoru poput povijesti parametara ili podataka o utrošenoj energiji.

.NET obuhvaća *ASP.NET i ASP.NET Core* te pruža okvir za pokretanje poslužiteljskih aplikacija. Okvir definira logički prostor u kojem aplikacija radi. Pri tome možemo konfigurirati i upravljati njezinim 'životnim ciklusom', podrazumijevajući pružanje svih potpornih alata i metoda za izvođenje programiranih zadataka. Životni vijek *TCP* veze jednak je 'životnom vijeku' same aplikacije.

Međuslojevi predstavljaju niz komponenata koje obrađuju zahtjeve i odgovore. Ove komponente obavljaju zadatke (poput upravljanja korisnicima aplikacije) usmjeravanja, vođenja zapisnika događaja i poziva metoda te drugih. Svaki međusloj ima specifičnu ulogu i može se dodati u cjevovod, stvarajući fleksibilnu arhitekturu za obradu zahtjeva.
Kada klijent šalje zahtjev poslužitelju, slijedi izvođenje protokola i metoda koja uključuju usmjeravanje, obradu zahtjeva i generiranje odgovora. Dolazi do faze usmjeravanja gdje se *URL* adresa zahtjeva uspoređuje s kontrolerom i akcijom kako bi se odredilo koje dijelove koda treba obraditi zahtjev.

Nakon toga zahtjev ulazi u cjevovod (*pipeline*) [21] međuslojeva. Svaki međusloj može izmijeniti zahtjev ili odgovor. Na primjer, međusloj za upravljanje korisnicima može provjeriti identitet korisnika te mu dodijeliti pristup određenim funkcionalnostima sustava, dok međusloj za vođenje dnevnika ili *log* datoteke bilježi informacije o prošlim zahtjevima. Koncept je sličan kao i u *OSI* modelu mreže gdje svaki međusloj vrši interakciju s podacima koji prolaze kroz njih. Kontroler je odgovoran za izvođenje aplikacijske logike i generiranje odgovora. Prikazi u *ASP.NET* predstavljaju *HTML* ili *XAML* [22] predloške koji se prikazuju na klijentskoj strani. Odgovor može sadržavati *HTML* sadržaj ili podatke u *JSON* formatu.

Postoje dvije verzije *.NET* okvira za razvoj poslužitelja: tradicionalni *ASP.NET* i moderni *ASP.NET Core* [23]. Verzija *ASP.NET Core* je preporučeni odabir za nove aplikacije. Jedna od prednosti *.NET frameworka* je mogućnost prilagodbe cjevovoda međuslojeva. Mogu se dodavati, uklanjati ili mijenjati komponente međuslojeva kako bi se zadovoljili specifični zahtjevi aplikacije. Primjerice, moguće je uključiti prilagođenu logiku za autentifikaciju, rukovanje iznimkama ili praćenje performansi kao dio međuslojeva. Fokus rada je na uspostavljanju baznoga sučelja i osnovnih naredbi. Svi servisi pokrenuti su na poslužitelju – serveru. Integrirana je *SQL* baza podataka kako bi bila omogućena trajna pohrana podataka o radu mikrokontrolera. Uz to, pogodnosti postojanja modela su višestruke. Za svaki novi regulator ili motor koji unosimo u sustav možemo koristiti jednostavnost sheme koju nam pruža ovaj *data* model. On je generaliziran te mu se parametri mogu namjestiti tako da funkcionira za sve tipove motora.

Upotreba modela pokazala se korisnom s obzirom na to da pruža recept izgradnje objekta u aplikaciji (regulatora ili motora) koji moramo ispoštovati prilikom razvoja. Konkretno, poslužiteljski dio aplikacije od mikrokontrolera preuzima podatke i obrađuje ih. Trenutna implementacija *TCP* komunikacije na mikrokontroleru za slanje koristi niz podataka, ili polje kao tip podataka, maksimalne veličine od 1.024 znakova.

Poslužitelj obrađuje dobivene podatke te ih sprema direktno u kreirani objekt klase *Motor.cs*. Svi neuspjelo dohvaćeni podatci postavljaju se na standardnu početnu vrijednost ili kao prazni. Tako kreiran objekt sada je pogodan za slanje i prikazivanje u bilo kojem dijelu aplikacije, na poziv s bilo koje strane ili sloja aplikacije. Na to je vezan i sam prikaz parametara pojedinoga uređaja. U

kodu grafičkoga sučelja koriste se petlje za ispitivanje svakoga parametra pojedinoga objekta putem refleksija [24]. Tako na jednostavan način dobivamo listu parametara koje možemo prikazati uz uporabu nekoliko linija koda. U nastavku je dan primjer *SQL* koda za kreiranje tablice koja se koristi za pohranu podataka, parametara i varijabli za definiciju motora. Prednost ovakve pohrane podataka je u tome što ih se može trajno pohraniti te analizirati povijest događanja i promjena nad podatcima.

```
CREATE TABLE MotorSim (
    -- Parametri stroja
    nominalVoltage INT,
    nominalSDCurrent INT,
    nominalPower FLOAT,
    nominalSQCurrent FLOAT,
    nominalSpeedRPM FLOAT,
    nominalSpeedRads FLOAT,
    nominalMoment FLOAT,
    Nf INT,
    Ns INT,
    Nra INT,
    permeab FLOAT,
    polePairs INT,
    J FLOAT,
    Ra FLOAT,
    La FLOAT,
    Rf FLOAT,
    Lf FLOAT,
    Laf FLOAT,
    c FLOAT,
    Ka FLOAT,
    Ta FLOAT,
    Kf FLOAT,
    Tf FLOAT,
    naz_tok FLOAT,
    km FLOAT,
    ke FLOAT
);

CREATE TABLE PowerInfo (
    DcBusVotage FLOAT,
    SQCurrent FLOAT,
    SDCurrent FLOAT,
    Time DateTime()
);

CREATE TABLE CommandData (
    startstop bool,
    setSpeed FLOAT,
    reverse FLOAT,
);
```

3. KOMUNIKACIJSKI SERVIS

Servis omogućava komunikaciju između aplikacije i mikrokontrolera te ostalih perifernih jedinica koje koristimo. On radi na principu komunikacije i obrade podataka u stvarnome vremenu te pruža niz funkcionalnosti ključnih za nadzor i upravljanje industrijskim postrojenjima. Jedna od funkcionalnosti ovoga servisa je sposobnost za komunikaciju putem *TCP (Transmission Control Protocol)* protokola, *MQTT* (*Message Queuing Telemetry Transport*) i *REST* (*Representational State Transfer*). To omogućava komunikaciju s pridruženim uređajima kojima upravljamo, omogućujući dvosmjernu razmjenu podataka. Primjerice, možemo poslati naredbe za kontrolu motora ili zahtjeve za prikupljanje senzorskih podataka. Moguće je pratiti brzinu vrtnje motora ili stanje senzora u stvarnome vremenu uz dojavljivanje svih promjena u dijelovima sustava kako bi se mogle izvršiti korekcije parametara za postizanje željenih kontroliranih vrijednosti.

Uspostavljanje veze između mikrokontrolera i servisa u sustavu temelji se na *TCP/IP* protokolu i slijedi osnovne principe komunikacije na različitim slojevima *OSI* modela. To se odvija korak po korak. Na fizičkom sloju događa se fizička veza između mikrokontrolera i *Sitara* servisa. Ovdje se razvija stvarna veza putem mrežnih uređaja, kao što su *Ethernet* sučelja. Sa strane pločice konfiguracija je napravljena u *Sysconfig* alatu, kako je ranije opisano. Ova se veza razvija indirektno korištenjem *TcpClient* klase. Konfiguracija *ethernet* protokola od strane klijenta vodi operacijski sustav sa svojim pridruženim upravljačkim programima za mrežnu karticu. Nakon što je fizička veza uspostavljena, mikrokontroler i servis moraju uspostaviti logičku vezu. Na ovome sloju događa se proces rukovanja. To je proces razmjene kontrolnih poruka između uređaja kako bi se dogovorili o parametrima komunikacije. U kodu se to događa kada se pozove metoda:

_tcpClient.Connect(_remoteEndPoint).

Mikrokontroler u ulozi klijenta šalje zahtjev za uspostavu veze prema odredišnoj IP adresi i portu na kojem poslužitelj očekuje zahtjeve. Primjer IP adrese s brojem porta izgleda ovako:

192.168.0.133:8888

gdje su adresa i broj *porta* odijeljeni dvotočkom. Budući da se komunikacija odvija preko usmjerivača igra ulogu međusobnoga povezivanja modula sustava pomoću *IP* adrese. Transportni sloj brine o pouzdanosti i integritetu podataka. *TCP* se koristi za ovu svrhu. *TcpClient* klasu te njezine metode upotrebljavamo za stvaranje *TCP* veze, što znači da će se koristiti *TCP* za prijenos podataka između mikrokontrolera i servisa. Nakon uspostave veze podatci se mogu prenositi bez potrebe za ponovnim uspostavljanjem veze. Podatci i naredbe se prema mikrokontroleru šalju putem metode:

_tcpClient.GetStream().Write(sendBytes, 0, sendBytes.Length);.

To su podatci koje mikrokontroler šalje prema poslužitelju za obradu. Na sličan se način podatci zaprimaju odgovarajućom metodom korištenjem klase _ *tcpClient.*
Servis čita podatke mikrokontrolera putem metode:

_tcpClient.GetStream().ReadAsync(receivedBytes, 0, receivedBytes.Length).

Kada su podatci primljeni, servis ih obrađuje i može odgovoriti na isti način. Kada je komunikacija završena, veza se može zatvoriti putem

_tcpClient.Dispose().

U procesu komunikacije između mikrokontrolera i servisa koristi se određeni tip podataka. Prije nego što se podatci šalju putem mreže, niz znakova tipa *char* pretvara se u niz bajtova tipa *byte array*. To se postiže korištenjem metode:

`Encoding.ASCII.GetBytes()`.

Svaki znak u znakovnom nizu pretvara se u odgovarajući *ASCII* broj i sprema u niz bajtova. Nakon što su podatci pretvoreni u niz bajtova, koristi se *TCP/IP* protokol za slanje ovoga niza bajtova

preko mreže prema odredišnom računalu, to jest prema poslužitelju. Servis prima ove podatke putem sljedeće naredbe:

await _tcpClient.GetStream().ReadAsync(receivedBytes, 0, receivedBytes.Length);.

Prilikom čitanja niz bajtova dekodira se i pretvara u odgovarajući *string* pomoću metode:

Encoding.ASCII.GetString().

Dobiveni se niz znakova potom filtrira kako bi se dobio odgovarajući tip podataka za daljnju obradu. Podatke koje želimo prikazati s mikrokontrolera pretvorili smo u JSON format. Nakon pretvorbe iz bajt formata u tekstualni oblik koristimo metodu

JsonSerializer.Deserialize()

pretvaramo zaprimljeni niz znakova u objekte i strukture kao što su

PowerInfo i CommandData.

Oni predstavljaju podatkovni model za slanje općih naredbi koje se naknadno mogu proširivati po želji programera te podatke poput trenutnih vrijednosti parametara. Ovaj proces osigurava uspješno slanje podataka iz mikrokontrolera do servisa gdje se podatci zatim obrađuju i koriste za daljnje proračune korištene u drugim dijelovima sustava. Važno je obratiti pažnju na ispravan format podataka kako bi se osigurala kompatibilnost međuslojeva aplikacije i pouzdana komunikacija među njima.

SignalR koristimo i na strani poslužitelja. Implementiran je kao čvor koji sadrži potrebne metode za komunikaciju s ostatkom sustava i povezuje se s čvorom na klijentskoj strani koji također ima prikladne metode za komunikaciju. Prilikom pokretanja aplikacije inicijalizira se objekt kreirane

klase *MotorHub*. Ovdje su sadržane metode, konfiguracija te inicijalizacija funkcionalnosti potrebnih za korištenje SignalR protokola.

Tu se također postavljaju potrebne veze i zavisnosti s drugim dijelovima sustava i programskim bibliotekama. Jedan od njih je i servis koji je odgovoran za komunikaciju s mikrokontrolerom. Servis i *MotorHub* klasa omogućuju da korisnik pomoću *MotorHub* klase putem *WebSocket* protokola (može koristiti i ostale protokole) šalje zahtjeve. Zadaća *MotorHub* klase, sa strane poslužitelja, je pozvati odgovarajuću metodu koja odgovara zahtjevu korisnika. Informacija se prosljeđuje sve do mikrokontrolera gdje se odvija konačno izvršavanje same naredbe. To podrazumijeva generiranje izlaznoga signala prema motoru.

Servis možemo smatrati posrednikom između korisnika i mikrokontrolera. Zadaća mu je preuzeti opterećenje izvođenja komunikacije mikrokontrolera prema mnoštvu korisnika te pružanja usluge regulacije parametara motora, uz minimalno proširenje i zagušenje postojećih resursa za izvođenje. Cilj je kreacija jednostavnog algoritma po *KISS* (*engl.* ”*Keep it simple, stupid”*) principu izrade. Primjeren je za izvođenje na ugradbenim sustavima čiji su resursi općenito limitirani naspram većih računalnih sustava sa željom izvršavanja algoritma u stvarnome vremenu.

Kada korisnik otvori aplikaciju i uspostavi vezu s poslužiteljem putem mrežnoga preglednika, *MotorHub* klasa pomoću svojih metoda detektira događaj te na nju reagira kako je definirano u metodi

OnConnectedAsync().

Ona korisniku šalje poruku o tome kako je veza s čvorom za komunikaciju uspostavljena putem indikatora na sučelju. *MotorHub* sadrži i ostale metode koje služe za upravljanje komunikacijom s mikrokontrolerom. Primjerice, metoda

ConnectToSitara()

pokreće uspostavu *TCP/IP* veze s mikrokontrolerom prema konfiguriranoj *IP* adresi i pripadajućim *portom*. Poslužitelj i mikrokontroler uspostavljaju vezu te ju drže 'živom' do upotrebe naredbe za prekidanje *TCP* veze. Kreirana je i metoda za pokretanje podatkovnoga niza

StartStream()
Ona pokreće kontinuirano slanje podataka s mikrokontrolera prema korisničkom sučelju. Metoda

StoreMotorParams()

služi za pohranu parametara koji se šalju mikrokontroleru. Detalji ovih parametara ovise o specifičnom projektu i njegovim zahtjevima. Ona služi kao univerzalna metoda za slanje bilo kojega parametra motora ili parametra regulatora s obzirom na to da se podatci u komunikacijskom cjevovodu (*pipeline*) prenose pretežito u *JSON* ili *Dictionary* formatu.
Metoda

DisconnectHubFromSitara()

koristi se za prekid veze između poslužitelja i mikrokontrolera. Ova metoda šalje odgovarajuću komandu mikrokontroleru putem servisa. Mikrokontroler prima naredbu i raskida se TCP veza.
Metoda

SendCommands()

omogućuje slanje specifičnih naredbi prema mikrokontroleru. Parametri naredbi mogu se prosljeđivati kao rječnik, ili engleski *Dictionary*, što omogućuje fleksibilnost u slanju različitih naredbi. *Dictionary* je tip podataka koji sadrži listu s uređenim parovima gdje sami možemo definirati tip pojedine komponente para. Na taj način je ostvarena identifikacija tipa zahtjeva koji mikrokontroler treba obraditi po dolasku zahtjeva od strane klijenta. Parovi su odvojeni dvotočkom te podsjećaju na *JSON* format koji smo spomenuli. On ima strukturu da su ključ i vrijednost odvojeni dvotočkom. Tako postoji par *a:0*, gdje ključ „a“ predstavlja naredbu prema mikrokontroleru, dok „0“ predstavlja vrijednost parametra ili ključa. Vrijednost je brojčanoga tipa te za svaki broj postoji lista pripadne naredbe koja se poziva. Vrijednost u primjeru predstavlja zahtjev za inicijalizacijom stalnoga toka podataka prema korisničkom sučelju. *MotorHub* klasa

također omogućuje ažuriranje korisničkoga sučelja s novim podatcima koji dolaze s mikrokontrolera. Metoda:

UpdateFrontend()

šalje te podatke korisničkom dijelu aplikacije putem *SignalR* protokola i ostalih pripadnih protokola.

Dependency Injection ili injektiranje zavisnosti praksa je koja se koristi u modernome razvoju softvera kako bi se olakšalo ponovno korištenje koda, održavanje i testiranje. U slučaju klase *MotorHub* injektiranje zavisnosti omogućuje da se u konstruktor klase šalje referenca na *Sitara* servis. Pod referencom podrazumijevamo povezivanje klasa i objekata adresama u memorijskom prostoru umjesto slanja cijeloga objekta ili kreiranja ostalih potrebnih objekata. Korištenjem ovakvoga pristupa omogućujemo da glavna klasa, koja poziva ili koristi neku vanjsku klasu koja joj je potreba za obavljanje funkcionalnosti, ne mora brinuti o specifičnoj implementaciji pojedinih funkcionalnosti, već je dovoljno proslijediti sučelje koje sadrži.

Klasa *MotorHub* odgovorna je za upravljanje komunikacijom između korisničkoga sučelja i poslužitelja servisa u stvarnome vremenu putem *SignalR* protokola. Ta klasa ne treba brinuti o detaljima komunikacije s mikrokontrolerom na nižoj, strojnoj razini. Za to su zaduženi upravljački programi strojne opreme i o njima brine operacijski sustav. Injektiranjem servisa postiže se princip pojedine odgovornosti (*Single Responsibility Principle*) koji omogućuje da svaka komponenta ima jasno definiran zadatak.

Dependency injection omogućava jednostavno zamjenjivanje stvarnih implementacija s *mock* ili testnim implementacijama tijekom testiranja jedinica. To znači da se funkcionalnost *MotorHub* klase može izolirano testirati pružajući *mock* model servisu, što je korisno za provjeru ponašanja aplikacije bez stvarne interakcije s mikrokontrolerom. Ubacivanjem ovisnosti klase *MotorHub* čini se fleksibilnijom i prilagodljivom budućim promjenama. Na primjer, ako se odlučimo promijeniti servis za komunikaciju s mikrokontrolerom, to je moguće učiniti bez mijenjanja klase *MotorHub*. Jednostavno zamijenimo injektirani servis novim.

SitaraService je komponenta programa odgovorna za interakciju s mikrokontrolerom. On se brine o detaljima komunikacije na nižim razinama, gledano prema strojnoj opremi, kao što su uspostava *TCP/IP* veze, slanje naredbi i primanje podataka s mikrokontrolera. Korištenjem servisa postižu se neke prednosti, poput ponovnoga korištenja koda. *SitaraService* klasa sadrži metode potrebne za komunikaciju s mikrokontrolerom. Ova se logika može ponovno koristiti u više dijelova aplikacije, ne samo u *MotorHub* klasi. Time se omogućava dosljedna i pouzdana komunikacija s mikrokontrolerom jer pomoću jedne provjerene metode zadužene za slanje podataka možemo obavljati pouzdanu komunikaciju za slanje širega spektra informacija i naredbi umjesto pisanja specifičnih metoda.

U slučaju nužnosti izmjene ili nadogradnje takvoga koda javljaju se problemi kada su pojedini moduli međusobno ovisni. To je u slučaju da jedan modul sadrži definicije koje su potrebne za rad drugoga modula. Takve promjene teško bi se pratile u smislu promoviranja promjena u svim dijelovima programskoga koda. U pristupu s injektiranjem ovisnosti putem konstruktora i korištenjem sučelja, sa svrhom odvajanja detalja implementacije od korisnika pristupa, postižemo laganu izmjenu koda na jednome mjestu.

Sučelja možemo zamisliti kao definirani skup vrijednosti i parametara koje možemo prilagođavati i na taj način upravljati sustavom koji se nalazi iza sučelja. U programskom smislu to znači da možemo napisati sučelje koje sadrži samo ime i tip metode te pripadne parametre koji se šalju u metodu. Klasa koja implementira definirano sučelje sadrži same definicije, to jest algoritam za izvođenje metode. U suštini, sučelje je popis metoda koje klasa posjeduje, a sama klasa koja implementira sučelje sadrži detaljan opis navedenih metoda. Za tako dizajniranu klasu i sučelje moguće je u više klasa injektirati isto sučelje te će svaka pojedina klasa imati osigurane sve metode navedene u sučelju. Prilikom samoga poziva metode iz sučelja referira se na baznu klasu na koju smo sučelje implementirali. Tako se prilikom izmjena na klasi u jednoj datoteci jamči da sva mjesta na kojima je injektirano sučelje imaju istu verziju koda.

Također, osigurava da je kod konzistentan jer sve metode definirane unutar sučelja moraju biti implementirane u metodi koja koristi to sučelje, inače kompajler javlja grešku. Odvajanjem logike komunikacije s mikrokontrolerom u odvojeni servis olakšava se upravljanje i održavanje koda.

Bilo kakve izmjene ili poboljšanja u procesu komunikacije s mikrokontrolerom mogu se provesti na jednome mjestu, što između ostaloga olakšava održavanje koda. *MotorHub* klasa ima zadaću korištenja *SignalR* i komunikaciju s korisničkim sučeljem dok se *SitaraService* servis usredotočuje na detalje komunikacije s mikrokontrolerom.

Datoteka *Program.cs* je konfiguracijska datoteka *.NET Core* ili *ASP.NET Core* aplikacije. To je ulazna točka za izvođenje aplikacije i sadrži *Main* metodu koja pokreće izvođenje programa. *Main* metoda u *Program.cs* datoteci je početna točka izvođenja aplikacije. Kad se pokrene aplikacija, operativni sustav poziva statičnu *Main* metodu, a iz nje se izvršava glavni tijek izvođenja. Ta metoda pripada *Program.cs* klasi i pomoću ključne riječi *static,* koju koristimo kao modifikator, klasu definiramo kao statičnu. Statične mogu biti i metode i klase. Statični član je identičan običnome članu, samo što njega nije moguće instancirati. Nije moguće koristiti operator *new* za stvaranje varijable tipa te klase. Budući da nema *instance* varijable, na članove statičke klase pristupa se korištenjem samoga imena klase.

U *Program.cs* datoteci se definiraju osnovne postavke kao što su portovi za slušanje, pristupne točke baze podataka, postavke autentifikacije i druge konfiguracijske opcije koje će aplikacija koristiti tijekom izvođenja. Poslužitelj je odgovoran za obradu zahtjeva, upravljanje komponentama i servisima i osigurava da aplikacija bude dostupna na mreži.

Ovo je mjesto gdje se obično dodaju i konfiguriraju servisi i komponente koje će se koristiti tijekom izvođenja aplikacije. To uključuje registraciju *Dependency Injection kontejnera* (spremnik sa servisima) kako bi se komponentama omogućio pristup servisima. Aplikacija ima definirane puteve do korištenih biblioteka i *URL* adrese u *Program.cs* klasi te se u njoj općenito postavljaju putevi do potrebnih resursa i mapiranje zahtjeva omogućujući aplikaciji da zna kako obraditi različite vrste zahtjeva. *Main* metoda pokreće poslužitelj servisa pomoću *Build().Run()* poziva. To započinje izvođenje aplikacije, pri čemu poslužitelj sluša dolazne zahtjeve i izvršava ih prema postavljenim konfiguracijama i rutama.

Proučene su različite komponente i usluge koje se koriste u *.NET* aplikacijama te kako pravilno konfigurirati njihov vijek trajanja. Ovo je bitno jer je indirektno vezano uz 'životni vijek' *TCP*

mrežne veze prilikom osvježenja aplikacije, između ostaloga. Usluge se mogu registrirati kao *singleton*, *scoped* ili *transient*, svaka s različitim vijekom trajanja.

Singleton usluge 'žive' tijekom cijeloga vijeka aplikacije i dijele se između svih zahtjeva. To je korisno za komponente koje trebaju globalno stanje, ali zahtijeva pažljivo upravljanje resursima i potencijalno može uzrokovati probleme ukoliko dođe do istovremenoga zahtjeva za istim resursima. Poželjno je održati *TCP* vezu 'živom' kroz čitav vijek trajanja aplikacije, sve do ručnoga prekida veze između mikrokontrolera i *servisa* pritiskom na *disconnect* gumb.

Scoped usluge imaju vijek trajanja ograničen na trajanje pojedinačnih zahtjeva. Ovo je korisno za komponente koje trebaju dijeliti stanje samo unutar jednoga zahtjeva i obično se koristi u aplikacijama koje koriste više niti.

Transient usluge stvaraju se na individualni zahtjev. Ovo je najpogodniji vijek trajanja za servise koji su bez stanja i nemaju podatke koji bi se trebali dijeliti između zahtjeva. Primjer ovoga su ostale veze prema i izvan servisa. Tako nema smisla kontinuirano dohvaćati podatke jedan po jedan, koji su već pohranjeni i postojeći u bazi podataka, jer ih se može dohvatiti kao postojeću listu. Zatim iz te liste izvući potrebno i ugasiti vezu prema *SQL* poslužitelju. U tom kontekstu, odgovara nam vijek *SQL* upita koji je postojeći samo za vrijeme izvršavanja upita.

Analizirani su problemi koji se mogu pojaviti ako se servisi ne konfiguriraju pravilno u kontekstu *Blazor* aplikacija. Pogreške poput "*Cannot provide a value for property 'HubConnection'*" i slične pojavljuju se ako servis nije registriran na odgovarajući način. Servisi unutar *Blazor* komponenata koriste se *HTTP* pozivima te metodama iz *SignalR* klase prema poslužitelju zahtjeva. Odluka o vijeku trajanja svake usluge ovisi o specifičnostima aplikacije i zahtjevima koje postavlja, a dobar dizajn servisa može znatno poboljšati performanse i održivost mrežne aplikacije. Datoteka

launchSettings.json

definira kako će se aplikacija pokrenuti i konfigurirati tijekom razvojnoga procesa. Datoteka pohranjuje konfiguracijske postavke za različite načine pokretanja *ASP.NET Core* aplikacije. *JSON* (*JavaScript Object Notation*) format je odabran zbog svoje jednostavnosti i čitljivosti. *JSON*

je format podataka koji koristi parove *ključ:vrijednost* i lako se raščlanjuje i generira, što ga čini pogodnim za konfiguraciju jer po strukturi i formatu odgovara modeliranju parametara sustava. Datoteku *launchSettings.json* koristi *Visual Studio* (ili drugi razvojni alati kao *Visual Studio Code*) kako bi konfigurirao načine pokretanja aplikacije za različite scenarije razvoja. U *launchSettings.json* mogu se definirati različiti profili za različite okoline.

Na primjer, može postojati poseban profil za razvoj, za testiranje i za produkcijsko okruženje. Svaki profil može definirati različite postavke kao što su adrese i veze, okolišne varijable i druge konfiguracijske parametre.

```
{
  "profiles": {
    "Development": {
      "commandName": "Project",
      "launchBrowser": true,
      "applicationUrl":
"https://localhost:5001;http://localhost:5000",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    },
    "Production": {
      "commandName": "Project",
      "launchBrowser": false,
      "applicationUrl": "https://production.example.com",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Production"
      }
    }
  }
}
```

U ovome primjeru postoje tri profila: *Development, Staging i Production*. Svaki profil definira različite *URL* adrese na kojima će se aplikacija pokrenuti, okolišne varijable i druge postavke.
Visual Studio automatski čita *launchSettings.json* i primjenjuje konfiguraciju prilikom pokretanja aplikacije u razvojnom okruženju. Na primjer, odabirom profila *Development* za pokretanje, aplikacija će se pokrenuti na *URL* adresama navedenim u tome profilu.
Ako specifični zahtjev za upravljanje električnim strojem zahtijeva veliku količinu zapisivanja u bazu, to može stvoriti zagušenje sustava i rezultirati velikim skupom podataka koje negdje treba pohraniti i koji mogu zasjeniti informaciju koju zapravo tražimo. Nakon odabira baze podataka, potrebno je konfigurirati vezu.

Korišten je *Microsoft SQL Server* za pružanje usluga baze podataka. To uključuje definiranje parametara kao što su *host*, *port*, korisničko ime, lozinka i ime baze podataka. Konfiguracija se generalno obavlja u datoteci (na primjer, *appsettings.json* u *ASP.NET Core*). U nastavku je dan isječak konfiguracijske datoteke s postavkama za spajanje na bazu podataka.

```
"ConnectionStrings":
{
   "DefaultConnection":    Server=Address;Database=DataBase;User=User;Password=Pass;"
}
```

ORM alati, poput *Entity Framework* za *.NET,* olakšavaju rad s bazom podataka tako da omogućavaju mapiranje objekata iz koda na tablice u bazi podataka. To pojednostavljuje upite i upravljanje podacima.

Sigurnost baze uključuje postavljanje dozvola na bazi podataka tako da samo autorizirani korisnici imaju pristup. Kako je nerealno da mikrokontroler podnosi mnoštvo zahtjeva, sustav je projektiran tako da je za distribuciju podataka prema van zadužen servis, u suradnji s bazom podataka gdje su spremljeni povijesni podatci te im se može pristupiti kao i u svakom redovnom slučaju korištenja *SQL* baze. Ona je dostupna zahtjevima većega broja korisnika te pruža oslonac u vidu memoriranja podataka što opet donosi višestruke pogodnosti za daljnje implementacije obrade podataka i slično.

3.1. Konfiguracija poslužitelja

U pozadini aplikacije, u *Program.cs* datoteci, registrirali smo potrebne servise za *SignalR* komunikaciju. To uključuje dodavanje potrebnih parametara u konfiguraciju datoteka servisa. Time je omogućeno poslužitelju da koristi *SignalR* za komunikaciju s korisničkim sučeljem i ujedno pruža podatke potrebne za crtanje i grafički prikaz parametara odnosno grafikona. Kreirana je *SignalR* klasa s nazivom *MotorHub*. Ovo čvorište služi kao centralno mjesto za komunikaciju između poslužitelja i korisničkoga sučelja. Definirane su metode koje klijenti mogu indirektno pozvati sa svrhom obrade poruka koje dolaze od mikrokontrolera. Na korisničkoj strani korišteno je injektiranje servisa za *IJSRuntime* [25] i *HubConnection* module.

IJSRuntime je sučelje koje omogućava komunikaciju s algoritmom i metodama za crtanje grafikona pomoću *JavaScript* skriptnoga jezika, dok je *HubConnection* klasa odgovorna za uspostavu veze s poslužiteljem, u ovome slučaju servisom za komunikaciju koji je izvor informacijskih paketa. Ovo je način kako aplikacija može pokrenuti komunikaciju s pozadinskim servisom. U slučaju zaprimanja paketa korišten je *IJSRuntime* za prikazivanje poruke o uspješnom pokretanju komunikacije ili o pogrešci. Slanjem podataka od mikrokontrolera prema korisničkom sučelju kontroler prvo prema servisu šalje paket podataka, a servis zatim preko *IJSR* obavještava, to jest pruža podatke za prikaz na grafikonima.
U metodi

OnInitializedAsync()

koja se poziva prilikom inicijalizacije komponente, najavili smo i registrirali metodu

Receive_MotorData()

za obradu poruka koje dolaze od servera putem *SignalR* protokola. Funkcionalnost se koristi za ažuriranje korisničkoga sučelja s novim podatcima koji dolaze od servera.

U aplikaciji su korišteni događaji (*events*) [26]. Događaji se koriste za hvatanje i reagiranje na korisničke akcije kao što su klikovi mišem, unos teksta ili bilo koja druga interakcija. U korisničkom dijelu aplikacije *HubConnection* klasa omogućuje povezivanje s čvorom s kojim želimo komunicirati pomoću napisanih metoda. Na strani poslužitelja, u čvorištu, smo definirali događaje (*events*) koji će biti okidači za komunikaciju. Na primjer, možemo imati događaj

MotorDataChanged

koji će se okinuti svaki put kada se podatci o motoru promjene. Na korisničkim stranicama pretplatili smo se na događaje pomoću metoda iz *HubConnection* klase. To znači da aplikacija treba pratiti taj događaj i na njega reagirati. Kada se događaj *MotorDataChanged* ostvari na poslužitelju (kada se podatci o motoru promjene), svi pretplatnici koji osluškuju događaj primaju

obavijest i izvršavaju prikladnu metodu. To može uključivati ažuriranje dobivenih podataka na ekranu, mijenjanje statusa veze ili mijenjanje stanja alarma u sustavu. Korištenjem ovoga mehanizma aplikaciji je omogućeno da na promjene reagira u stvarnome vremenu.

Ako se podaci o motoru promijene, ti podaci će automatski stići do svih klijenata koji su pretplaćeni na događaj *MotorDataChanged*, što omogućava brzu i dinamičnu aktualizaciju korisničkoga sučelja. U ovome kontekstu, na klijente se misli na sve dijelove koda koji su pretplaćeni na specifičan događaj, to jest korisnici metoda koje pruža *EventHandler* klasa. Ovo je intuitivan način za implementaciju stvarnoga vremena u mrežnim aplikacijama jer omogućava komunikaciju između poslužitelja i klijenata, bez potrebe za stalnim osvježavanjem stranice ili ručnim interakcijama korisnika.

Prilikom slanja zahtjeva prema mikrokontroleru vodilo se računa o uspostavljanju stalnoga kanala koji šalje podatke. Od mikrokontrolera možemo zahtijevati više operacija paralelno, slanje podataka s motora te mogućnost primanja naredbi za upravljanje motorom i konfiguraciju pojedinih dijelova i parametara sustava. Iz toga razloga bitno je spomenuti koncepte paralelnog izvođenja. Takvo ponašanje postižemo upotrebom *Taskova* i *Threadova*.

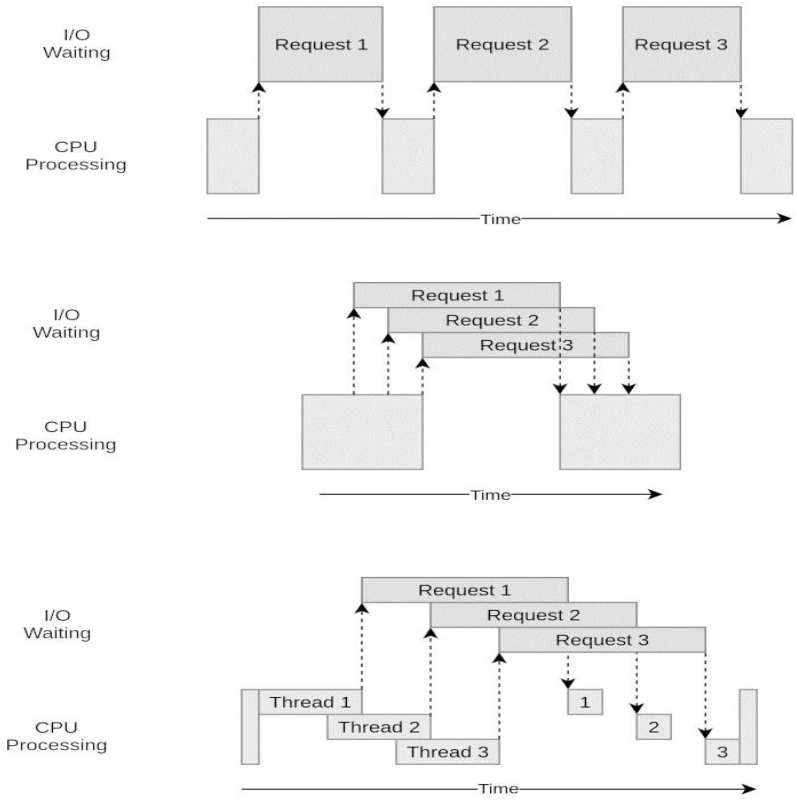
Async Task i *Thread* [27] su različiti mehanizmi za izvođenje paralelnih operacija u programiranju. Svaki ima svoje specifične karakteristike i namjene. *Async Task* predstavlja višu razinu apstrakcije i omogućuje upravljanje asinkronim operacijama na jednostavniji način. Najčešće se koristi za obradu I/O operacija i smanjuje potrebu za stvaranjem i upravljanjem dodatnim nitima. *Async Task* omogućuje pisanje koda koji se izvodi sekvencijalno, ali istovremeno izvodi asinkrone operacije. Ovo je posebno korisno za operacije koje čekaju vanjske resurse, kao što su čitanje iz datoteke ili slanje mrežnih zahtjeva.

S druge strane, *Thread* predstavlja nižu razinu apstrakcije i omogućuje izravno stvaranje i upravljanje nitima u programu. Ovo je korisno za operacije čije je izvođenje opterećenje za procesor i koje mogu iskoristiti paralelno izvršavanje na više jezgri procesora. Stvaranje previše niti može dovesti do problema s resursima i performansama. *Async Taskovi* su obično izvedeni na jednoj niti i koriste mehanizme poput petlje ili bazena niti za upravljanje asinkronim operacijama.

Threadovi, s druge strane, omogućuju stvarno izvođenje pomoću više paralelnih niti izvođenja i mogu se izvoditi paralelno na više jezgara procesora. Kada koristimo *Async Task*, sinkronizacija i upravljanje blokovima koji više nisu aktivni izvršava se automatski implementiranim metodama unutar pripadnih klasa i biblioteka. Ovo je važno jer su sinkronizacijski problemi česti u programima s više niti izvođenja i njihovo manualno rješavanje može biti izazovno. Kada koristimo *threading*, programer je odgovoran za ručno upravljanje sinkronizacijom.

Što se tiče performansi, *Async Task* je često učinkovitiji za I/O operacije jer izbjegava, u vidu resursa, troškove stvaranja i upravljanja nitima. Pristup pri korištenju niti je bolji za operacije koje se izvode na procesoru, ali treba voditi računa o njihovom broju i sinkronizaciji kako bi se izbjegli problemi sa skaliranjem i resursima.

Izbor između *Async Task* i *Thread* opcija ovisi o specifičnim potrebama programa. Ako postoji dio sustava s pretežito I/O operacijama i želimo jednostavno upravljati asinkronim zadacima, *Async Taskovi* su dobar izbor. Ako su potrebne operacije koje intenzivno koriste procesor s mogućnošću paralelnog izvršavanja, tada su *Threadovi* prikladniji, ali zahtijevaju pažljivo upravljanje. U našem slučaju iskoristili smo *Threading* tehniku zato što želimo eksplicitno stvoriti novu, paralelnu nit izvođenja s posebnim setom dodijeljenih računalnih resursa kako bi se postiglo potpuno izolirano izvođenje od glavne niti izvođenja. Na ovaj način nit možemo prekinuti pomoću *CancellationToken* podatkovne strukture koju šaljemo pritiskom na odgovarajući gumb na korisničkom sučelju, čime se mijenja svojstvo tokena za otkazivanje zadatka i nit koja izvršava slanje podataka je terminirana. Pošto se zadatak pokreće i izvodi na posebnoj niti izvođenja, to dopušta glavnoj programskoj niti da bez blokiranja obavlja ostale zadatke za koji je sustav namijenjen.



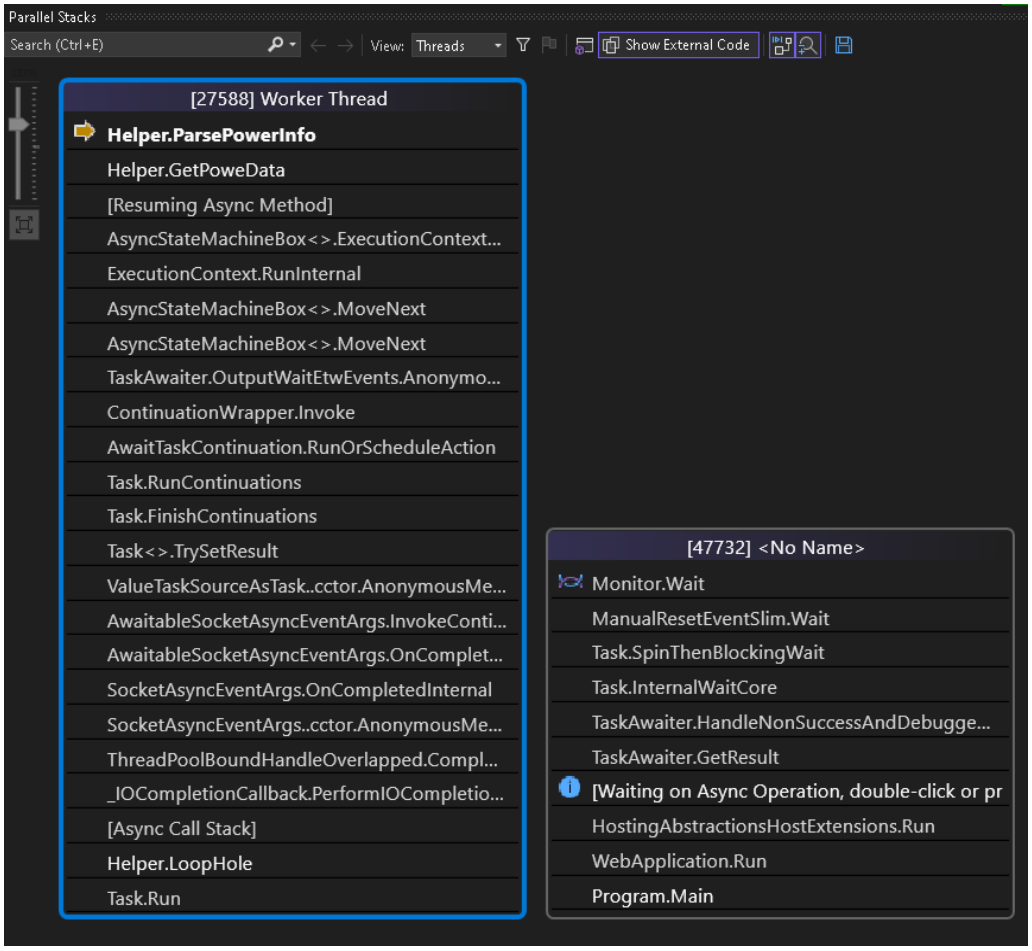
Slika 3. Usporedba Taska (I/O bound) i Threada (CPU bound) [28]

Na Slici 3. vidimo tri slučaja za svaki način izvođenja. U prvome slučaju imamo jednu nit izvođenja s više zahtjeva za čitanje i pisanje. U standardnom izvođenju za svaki zahtjev moramo prekinuti izvođenje te obraditi I/O zahtjev. U drugome slučaju imamo jednu programsku nit, no zahtjevi iz prvoga slučaja izvode se u načinu izvođenja koji ne blokira ostale procese. To jest, sustav dopušta izvođenje glavne petlje koja je inicijalizirala upotrebu niti unatoč čekanju na I/O operaciju. Na trećoj vremenskoj crti prikazano je izvođenje više niti paralelno.

Tako zaključujemo da postoje posebni resursi za svaki *thread* te se, na primjer, *thread 2* može izvoditi za vrijeme izvođenja I/O zadaće za *thread 1*. Upravo ovakvo ponašanje poželjno je u sustavu kako bi se I/O operacije jednoga zadatka mogle odvijati nezavisno od I/O operacije drugoga zadatka, pri čemu se u ovome slučaju prvi zadatak odnosi na slanje podataka prema korisničkom sučelju, a drugi na obrađivanje zaprimljenih podataka. Sustav je modeliran na način da se, koliko je moguće, kalkulacijsko opterećenje prenosi na poslužitelja usluge naspram

mikrokontrolera. To je da se rastereti mikrokontroler od nepotrebnih operacija, pošto posjeduje puno manje resursa od poslužitelja, u vidu procesorske snage i dostupne radne memorije.

Na sljedećoj slici (Slika 4.) prikazan je stog s trenutnim nitima izvođenja. Izvođenje programa zaustavljeno je točno u trenutku slanja zahtjeva za kontinuiranim slanjem podataka s mikrokontrolera prema korisniku. Programski kod pripada poslužitelju servisa. Točka za zaustavljanje izvođenja nalazi se u pomoćnoj metodi za izvlačenje relevantnih informacija iz zaprimljenoga podatkovnog paketa.



Slika 4. Paralelno izvođenje glavne niti aplikacije i niti za dohvaćanje podataka

Tako vidimo da je izvođenje stalo na izvođenju metode

ParsePowerInfo()

u pomoćnoj klasi *Helper*.

Helper klasa kreirana je kao pomoćna klasa čije se metode izvode u odvojenoj niti izvođenja. Na taj način zaprimanje podataka odvojeno je od glavne niti izvođenja programa te se te operacije izvršavaju paralelno, bez da funkcija kontinuiranoga prikupljanja informacija svojim radom blokira izvršavanje ostatka programa. Za to vrijeme glavna nit izvođenja programa čeka na zaprimanje novih podataka, ali istovremeno može nastaviti dalje s izvođenjem prikaza podataka i primanjem ostalih naredbi.

4. AM2634 MIKROKONTROLER

Mikrokontroler upravlja radom elektroničkoga energetskoga pretvarača na koji je spojen stroj istovremeno prikupljajući podatke mjerenja (struje, naponi, brzina vrtnje…). U nastavku ćemo detaljnije upoznati njegove funkcionalnosti i ulogu u elektromotornom pogonu. U mikrokontroler je implementirana upravljačka struktura pretvarača prilagođena za upravljanje asinkronim strojem primjenom vektorskog upravljanja orijentacijom toka. Na primjer, ako želimo da se motor okreće određenom brzinom ili u određenom smjeru, mikrokontroler prima naredbe od poslužitelja putem mrežne komunikacije. Zatim pomoću regulacijske strukture generira prikladan *PWM* signal pogodan za upravljanje sklopkama izmjenjivača koje stroju daju energiju za rad, pritom energija dolazi iz električne mreže na koju je stroj spojen.

Osim upravljanja motorima, mikrokontroleri imaju senzore i mjerne uređaje koji prikupljaju vrijednosti parametara sustava. To mogu biti snimani podatci o temperaturi, tlaku, brzini vrtnje, struji te ostali parametri. Redovito šalju prikupljene podatke aplikaciji kako bi se informacije mogle obraditi ili prikazati korisniku. Primaju upute i naredbe iz aplikacije kako bi se prilagodio rad motora.

Prikupljanje podataka svodi se na očitavanje mjerenih veličina električnoga stroja u željenom trenutku, kao što su vrijednosti brzine vrtnje, struje i napona u pojedinim granama stroja (d i q osi stroja). Podatke možemo poslati i u vidu zadanih vrijednosti prema regulacijskoj strukturi. Vrijede slična pravila izmjene podataka kao i kod *SignalR* koji također koristi *TCP* u pozadini, uz razliku da ovdje koristimo samostalni *TCP* protokol, bez dodavanja protokola viših slojeva. Protokoli na nižim slojevima mreže korišteni su pošto su neophodni za uspostavljanje komunikacije s mikrokontrolerom. Kao i kod veze između poslužitelja i korisničke aplikacije, tako je i veza između poslužitelja i mikrokontrolera stalna.

Međutim, unatoč svojoj pouzdanosti, *TCP* u *.NET* može pokazati neka ograničenja kada se suoči s visokim opterećenjem i brzim slanjem velikoga broja malih paketa [29]. Kao takav se ne može koristiti za komunikaciju u stvarnome vremenu jer nema zagarantiranu brzinu prijenosa poput industrijskih standarda poput *Profibus* ili *Ethercat*. Ograničenje može rezultirati određenim kašnjenjem ili zadrškom u isporuci paketa, često u rasponu od 100 do 200 milisekundi ili čak i

više. U uputama promatrane klase koju koristimo za komunikaciju dani su prijedlozi za pristup dizajnu aplikacije u takvim slučajevima, poput generiranja većega tijela poruke, skupljanja podataka u listu te slanje liste umjesto individualnih elemenata.
Razlog ove pojave je u načinu na koji *TCP* upravlja paketima i kako se rukuje zagušenjima u mreži. Protokol koristi algoritme za kontrolu protoka kako bi se osiguralo da mreža ne bude preopterećena i da se izbjegnu kolizije paketa. Kada se brzo šalju velike količine malih paketa, *TCP* može reagirati na nekoliko načina koji dovode do zadrške.

TCP prijemnik mora privremeno pohraniti pristigle pakete dok ne budu spremni za obradu. Kada je velik broj malih paketa poslan brzo, to može dovesti do zagušenja i povećanja vremena potrebnoga za obradu paketa. *TCP* koristi algoritme za kontrolu zagušenja kako bi prilagodio brzinu slanja paketa prema trenutnim uvjetima mreže. Kada se zagušenje dogodi, *TCP* će privremeno usporiti slanje paketa, što rezultira kašnjenjem. Ovo kašnjenje je varijabilno i predstavlja ograničavajući faktor za korištenje ovog tipa protokola u sustavima koji zahtijevaju zagarantiranu brzinu prijenosa podatka. Mali paketi se često segmentiraju kako bi se prilagodili maksimalnim *MTU-u* (*Maximum Transmission Unit*) veličinama mrežnih slojeva. Ovo može dodatno povećati broj paketa i utjecati na brzinu isporuke. Umjesto slanja velikoga broja malih paketa, razmotreno je slanje manjega broja većih paketa kako bi se smanjilo opterećenje mreže i međuspremnik na strani prijemnika. Optimizacija protokola postiže se prilagodbom parametra *TCP* protokola kako bi bolje odgovarali potrebama aplikacije. To uključuje podešavanje algoritama za kontrolu zagušenja.

Korištenje *UDP* protokola je razmatrano i isprobano pošto po svojoj prirodi nema spomenutu zadršku poput *TCP*-a. *UDP* ne nudi pouzdanu isporuku kao *TCP*, ali je brži i bolje se prilagođava aplikacijama koje zahtijevaju brzu i stvarnu vremensku komunikaciju. No u razvoju aplikacije odlučeno je koristiti sigurno i pouzdano slanje i primanje podataka naspram upotrebe jednostavnije i naizgled lakše metode.

4.1. Komunikacijska mreža i povezivanje dijelova sustava

Integracija AM2634 mikrokontrolera započela je s postojećim projektom koji je koristio *TCP/IP* za komunikaciju. Prvobitno je ovaj projekt bio dizajniran za povremenu komunikaciju s mikrokontrolerima gdje bi se mikrokontroleri povezivali s centralnim sustavom samo po potrebi. Međutim, kako bismo omogućili stalno praćenje i upravljanje, bilo je potrebno izmijeniti logiku djelovanja postojećeg algoritma. Za početak, navedene su korištene biblioteke čije smo funkcije koristili pri obradi podataka.

```
#include <string.h>
#include <stdio.h>
#include "lwip/opt.h"
#include "lwip/sys.h"
#include "lwip/api.h"
#include <kernel/dpl/ClockP.h>
#include "enet_apputils.h"
#include "FreeRTOS.h"
#include <math.h>
#include <kernel/dpl/DebugP.h>
#include <kernel/dpl/CycleCounterP.h>
#include <mathlib/trig/ti_arm_trig.h>
#include "ti_drivers_open_close.h"
#include "ti_board_open_close.h"
```

AppTcp_ServerTask() je funkcija koja predstavlja poslužiteljski zadatak. Ovdje se definiraju potrebne varijable, uključujući objekte za upravljanje vezom s klijentom. Također se postavljaju i konfiguriraju različiti parametri veze, uključujući *port* te *IP* adrese. Ovo je jedina nit izvođenja na jezgri zaduženoj za komunikaciju kako ne bi bilo zastoja ili kašnjenja podataka između klijenta i motora.

```
static void AppTcp_ServerTask(void *arg)
{
    struct netconn *pConn = NULL, *pClientConn = NULL;
    pConn = netconn_new(NETCONN_TCP);
    netconn_bind(pConn, IP_ADDR_ANY, APP_SERVER_PORT);
    ...
}
```

Unutar petlje provjerava se je li stigao paket i ako jest podatci iz paketa se čitaju i šalju prema klijentu. Ovo je, naravno, ovisno o vrsti zaprimljenoga paketa, to jest kakav je točno zahtjev klijent poslao na server. Tako razlikujemo naredbe poput dohvati podatke koje *TCP* poslužiteljski

algoritam izvršava na jezgri mikrokontrolera nakon zaprimanja naredbe odmah procesira te počinje slati pakete natrag klijentu.

Slanje se izvršava u definiranome vremenskom periodu te obuhvaća sve nove promjene podataka unutar regulacijske strukture koje su se dogodile netom prije slanja podataka. Isto tako, možemo poslati naredbu za spremanje parametara motora ili regulatora, pri čemu će *TCP* poslužitelj (mikrokontroler) poslane informacije obraditi i spremiti unutar memorije regulacijske strukture koja se nalazi u memoriji mikrokontrolera.

Prije toga potrebno je inicijalizirati te ostvariti *TCP* vezu između klijenta i servisa. U ovome kontekstu servis za komunikaciju s mikrokontrolerom koji smo realizirali pomoću *.NET* smatra se klijentom, dok je sam mikrokontroler *TCP* poslužitelj. Najprije kreiramo potrebne spremnike podataka te čekamo da se klijent spoji. Nakon spajanja klijenta, vršimo dodatne provjere stanja veze te ukoliko nema zastoja ili grešaka sustav prelazi u sljedeći korak algoritma.

```
//TCP SERVER CODE
/* Tell connection to go into listening mode. */
   netconn_listen(pConn);
    while (1){
        struct netbuf *buf;
        void *receivedData;
        u16_t receivedLen;
        char timeString[20];
        printf("?  Awaiting new connection  ??\r\n");
        err = netconn_accept(pConn, &pClientConn);
        //netconn_set_nonblocking(conn, );
        printf("!  Accepted new connection  ! \r\n ConnectionID: %p\r\n", pClientConn);
        if (err < ERR_OK)
        {
            DebugP_log("ERR Connection: errno %d\r\n", err);
            continue;
        }
        size_t dataLen ;
  …}
```

Sljedeći korak podrazumijeva samo slušanje zahtjeva od strane poslužitelja. Ideja je da mikrokontroler u beskonačnoj petlji stalno sluša zahtjeve te na njih odgovara prikladnim odgovorom. Taj algoritam možemo rastaviti u sljedećih nekoliko koraka. Petlja čeka dolazak novih zahtjeva klijenata. To se postiže pozivom funkcije

netconn_recv(pClientConn, &buf).

Ova funkcija blokira izvođenje dok ne stigne zahtjev ili dok se ne pojavi neka pogreška. Kad stigne zahtjev klijenta, petlja čita podatke iz toga zahtjeva i sprema ih u međuspremnik. Zatim se podaci iz spremnika čitaju i kopiraju u varijablu koja sprema podatke iz zahtjeva. Nakon što se podaci iz zahtjeva obrade, petlja šalje odgovor klijentu. To se postiže pozivom funkcije

netconn_write(pClientConn, data, len, NETCONN_COPY)

gdje se *data* odnosi na podatke koji su prethodno pročitani iz zahtjeva, a *len* je dužina tih podataka. Podatci koji su ovdje sadržani dolaze direktno preko međuprocesne komunikacije ili *IPC-a*. Diskretna regulacijska struktura tijekom svoga rada sve bitne podatke o radu motora može pohraniti ili predati jezgri zaduženoj za komunikaciju s poslužiteljem. U nastavku je dan primjer korištenoga koda u *Code Composer Studio* okruženju. Ovaj dio koda procesira pojedine naredbe zaprimljene od strane poslužitelja. U nastavku su primjeri obrade zaprimljenoga zahtjeva, u dva dijela. Prvi dio opisuje kontinurano slanje podataka te spremanje pojedinoga parametra motora.

```
if (strcmp((char *)receivedData, "a:0") == 0)
{
   DebugP_log("..[!]Received command a:0--> just send data\r\n");
   sys_thread_new("justSendData", justSendData, newClientConnection,
DEFAULT_THREAD_STACKSIZE,                              DEFAULT_THREAD_PRIO);
}

//   Save Motor Params
else if (strncmp((char *)receivedData, "a:1",3) == 0)
{
   DebugP_log("..[!]Received command a:1--> saving motor parameters to  DSP...\r\n");
   DebugP_log("Parameters to save:%s\r\n", receivedData+=3);
   commandPacket = (CommandPacket){2, "KPje1\r\n"};
   RPMessage_send( &commandPacket, sizeof(commandPacket),  INVERTER_CORE_ID,
INVERTER_ENDPOINT,RPMessage_getLocalEndPt(&gAckReplyMsgObject), 100);
}
```

Nastavak s ostalim naredbama:

```
//   Set Ref speed
else if (strncmp((char *)receivedData, "b:3",3) == 0)
{
   DebugP_log("..[!]Received command a:3--> set ref speed[%s] @ rpm\r\n",
receivedData);
   commandPacket = (CommandPacket){2, receivedData};
   RPMessage_send(&commandPacket, sizeof(commandPacket),  INVERTER_CORE_ID,
INVERTER_ENDPOINT,  RPMessage_getLocalEndPt(&gAckReplyMsgObject), 100);
}
else if (strcmp((char *)receivedData, "a:-2") == 0)
{
   DebugP_log("..[!]Received command a:-2--> disconnecting from DSP...\r\n");
}
else if (strcmp((char *)receivedData, "b:1") == 0)
{
   DebugP_log("..[!]Received command b:1 --> Starting Motor..\r\n");
   commandPacket = (CommandPacket){1, "ON"};
   RPMessage_send(&commandPacket, sizeof(commandPacket),
                        INVERTER_CORE_ID, INVERTER_ENDPOINT,
                        RPMessage_getLocalEndPt(&gAckReplyMsgObject),
                        50);
}
else if (strcmp((char *)receivedData, "b:0") == 0) {
   DebugP_log("..[!]Received command b:0 --> Stopping Motor..\r\n");
   commandPacket = (CommandPacket){1, "OFF"};
   RPMessage_send(
                        &commandPacket, sizeof(commandPacket),
                        INVERTER_CORE_ID, INVERTER_ENDPOINT,
                        RPMessage_getLocalEndPt(&gAckReplyMsgObject),
                        50);
                }
else if (strcmp((char *)receivedData, "b:2") == 0)
{
   memset(dataToSend, 0, sizeof(dataToSend));
   dataLen = strlen(dataToSend);
   if (dataLen <= 0 || dataLen >= 700){DebugP_log("Data length is
invalid\r\n");continue;}
   err = netconn_write(pClientConn, dataToSend, dataLen, NETCONN_COPY);
   if (err != ERR_OK){DebugP_log("Failed to send data: %d\r\n", err); continue;}
}
//   Unknown command
else
{
   DebugP_log("..[!]Unknown command: %s\r\n", (char *)receivedData);
}
```

Nakon što se odgovor pošalje klijentu, petlja provjerava ima li još podataka za obraditi. Ako ima, ponavlja se ciklus obrade zahtjeva. Ako ne, petlja čeka novi zahtjev ili eventualno prekida obradu ako dođe do pogreške. Ideja je da na bilo koji znak prekida veze između poslužitelja i mikrokontrolera upravljački program reagira na način da odbacuje trenutnu vezu te ponovno ulazi u stanje čekanja nove *TCP* veze putem vanjske beskonačne petlje u koju ulazimo direktivom *continue* prilikom promjene statusa veze u različito od spojeno.

U slučaju bilo kakve pogreške pri komunikaciji s klijentom ili ako veza više nije valjana, petlja prekida obradu i izlazi iz nje. To se događa kroz promatranje stanja varijable koja sadrži status komunikacije s klijentom. Naravno, u fazi razvoja moguće je ispisivati pogrešku u konzolu kako bi se vidjelo o kojoj se pogreški radi i kako bi se olakšalo njezino ispravljanje.

Korištenjem funkcionalnosti *TCP* poslužitelja, mikrokontroler je spreman prihvaćati veze i obraditi dolazne podatke kad god je to potrebno. Nakon konfiguracije mikrokontroler koristi funkciju *netconn_listen* kako bi dojavio poslužitelju da čeka dolazne veze. Ova funkcija čeka i prihvaća nove veze klijenata. Promotrimo sljedeću funkciju:

*err_t netconn_listen(struct netconn *conn);*.

Parametar **conn* je pokazivač na strukturu koja opisuje mrežnu vezu i tipa je *struct netconn* te sadrži informacije i parametre, uz konfigurirane modele potrebne za rad sustava za izmjenu poruka. Na mikrokontroler smo postavili naredbu kojom želimo postaviti jezgru na kojoj se izvodi ovaj algoritam u stanje slušanja novih podataka korištenjem spomenute strukture. To znači da ta struktura u sebi ima zapisano koji *port* i koji *IP* mora koristiti da bi uspješno primio poruke koje na taj *port* i *IP* dolaze, naravno uz ostale parametre veze koje smo konfigurirali u ostalim koracima. Kada je sve pravilno konfigurirano, mikrokontroler je u mogućnosti osluškivati dolazne veze i podatke s istih.

Funkcija *netconn_listen* vraća *ERR_OK* vrijednost ako je uspješno postavljena mrežna veza prema poslužitelju. Vrijednost koja se vraća je brojčanoga tipa i naknadno se preslikava na odgovarajuću tekstualnu vrijednost poruke. U slučaju problema, vraća odgovarajuću pogrešku. Kada se pozove funkcija *netconn_listen*, uređaj prelazi u način osluškivanja veze. To znači da će mikrokontroler sada prihvaćati dolazne poruke na određenom *portu* (definiranom prilikom stvaranja veze) i omogućit će interakciju s klijentima.

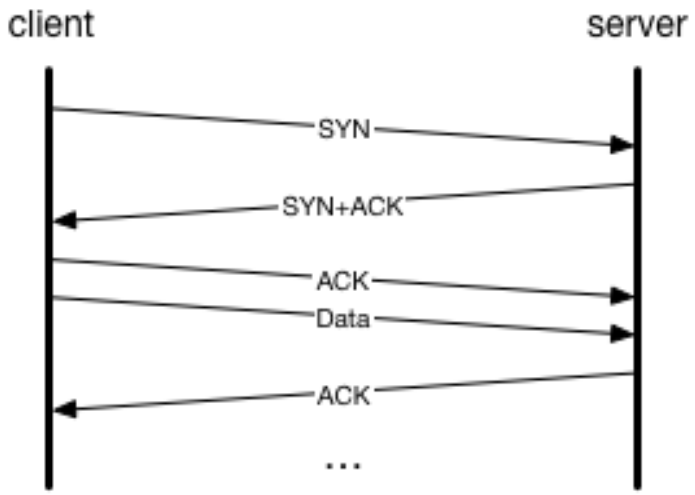
TCP proces rukovanja između klijenta i poslužitelja omogućava siguran prijenos podataka između pošiljatelja i primatelja. Sinkronizacijski korak ili *SYN* počinje kada računalni uređaj koji želi uspostaviti vezu (pošiljatelj) šalje prvu *TCP* poruku poznatu kao *SYN* poruka. Pošiljatelj također

odabire prikladan broj vrata ili *port* na kojem će primatelj prihvaćati poruke. Odgovor na sinkronizacijski korak daje primatelj koji također želi uspostaviti vezu te zaprima *SYN* poruku i na nju odgovara.

Primatelj šalje *SYN-ACK* poruku i potvrđuje da je zaprimio *SYN* poruku. Primatelj također bira prikladni *port*. Nakon što pošiljatelj primi *SYN-ACK* poruku, šalje potvrdu ili *ACK* potvrdu natrag primatelju. *ACK* potvrda obavještava primatelja da je veza uspostavljena i da su spremni za razmjenu podataka. Nakon uspostave veze, obje strane mogu sigurno slati i primati podatke, s potvrdom o primitku čime garantiramo dosljedan prijenos podataka.

Ovaj postupak koji se odvija u tri faze osigurava dvosmjernu komunikaciju između uređaja. Pošiljatelj i primatelj razmjenjuju početne sekvencijske brojeve kako bi se osiguralo da su poruke koje pristižu ispravno posložene i da nema pogrešaka u komunikaciji. *TCP* rukovanje također osigurava pouzdanu vezu i obrambene mehanizme protiv paketa koji bi mogli biti izgubljeni, duplicirani ili iskrivljeni tijekom prijenosa.

Nakon uspješnoga završetka *TCP* rukovanja, uređaji mogu kontinuirano razmjenjivati podatke tijekom cijele veze. Kada komunikacija završi, koristi se poseban postupak zatvaranja veze kako bi se oslobodili resursi i osiguralo pravilno uklanjanje korištenih resursa (npr. kako bi se oslobodio korišteni *port* za nove korisnike). Taj postupak je veoma sličan postupku uspostavljanje veze, no, za primjer, umjesto *SYN* poruke šalje se *FIN* poruka.



Slika 6. Proces TCP rukovanja [30]

Prilikom rada *TCP* servera na mikrokontroleru ideja arhitekture te upravljanje nitima i procesima je sljedeća. Želimo da postoji samo jedna veza između mikrokontrolera i poslužitelja. Poslužitelj obrađuje sve daljnje vanjske zahtjeve te samo on u vezi jedan na jedan može komunicirati s mikrokontrolerom. To znači da želimo uspostaviti trajnu *TCP* vezu između mikrokontrolera i poslužitelja na zahtjev te ju prekinuti na eksplicitni zahtjev korisnika. Kada se ta veza uspostavi, držimo ju 'živom' i u njoj beskonačno puta očekujemo zahtjeve korisnika, gledano iz perspektive mikrokontrolera, *TCP Servera*, koji je pokrenut na njemu.

To je iz razloga jer *TCP* protokol sam po sebi, naspram *UDP* protokola, koristi rukovanja, što samo po sebi ima neko mjerljivo vrijeme izvođenja te posljedično unosi nepotrebne zastoje. *TCP* poslužitelju je dodijeljena individualna jezgra na procesoru mikrokontrolera te je cijela jezgra posvećena obavljanju zadataka, to jest niti izvođenja u slučaju *FreeRTOS* izvedbe te upotrebe dretvi. Pošto poslužitelj u ovakvoj arhitekturi nema opterećenja oko instanciranja velikoga broja veza te posljedično i puno memorije te procesor koji je fokusiran isključivo na izvođenje operacija u stvarnome vremenu, osiguravamo maksimalnu efikasnost podjelom zadataka na one dijelove sustava koji su svojom funkcionalnošću za isti i namijenjeni. Tako je *TCP* poslužitelj pokrenut na sekundarnoj jezgri mikrokontrolera i zadužen je za protok podataka s minimalno kašnjenja, od regulacijske strukture prema korisničkom sučelju, preko poslužitelja.

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*",
  "ApiSettings": {
    "BaseUrl": "https://0.0.0.0:8080"
  },
  "ConnectionStrings": {
    "DefaultConnection": "Server=localhost;Database=Ritroller;User
Id=***;Password=***;"
  }
}
```

U postavkama prikazanima u gornjem konfiguracijskom kodu smo definirali adresu na kojoj se nalazi poslužitelj i adresu pristupne točke na koju se mogu spojiti klijenti. Također, ovdje su konfigurirane i postavke za spajanje na bazu podataka. Treba voditi računa o dodjeljivanju svih ostalih korištenih resursa poput perifernoga dijela sustava za ispis i učitavanje podataka te upotrebu ispisa u konzolu aplikacije. Takve operacije mogu često trajati mnogo dulje, reda veličine sto ili tisuću puta, ovisno o veličini i učestalosti ispisa. Upis varijable u memorijsku lokaciju može trajati par ciklusa izvođenja što mjerimo u vremenskim periodima reda mikrosekunda i nanosekunda, dok ispis u konzolu može potrajati i do sekunde. U tom slučaju će proces ispisa, koji nije kritičan za izvođenje operacije, svojim vremenom izvođenja nadmašiti vrijeme slanja podatkovnih paketa.

Drugim riječima, sekvencijalno izvođenje naredbe za slanje ili primanje paketa praćeno ispisom podataka u konzolu uzrokovat će vrijeme izvođenja ukupnoga trajanja kao zbroja trajanja samostalne naredbe s dodanim vremenom ispisa u konzolu. Zbog toga će doći do desinkronizacije s intervalom vremena čitanja podataka na korisničkoj strani što bi moglo dovesti do pogrešne interpretacije podataka. Ispis u konzolu može se i izbjeći, no želja je pokazati da općenito prilikom korištenja računalnih resursa, pogotovo u scenariju izvođenja u stvarnome vremenu, treba paziti na svaki detalj, to jest minimizirati upotrebu nepotrebnih resursa.

API, ili sučelje za interakciju s pojedinim sustavom, generalno predstavlja set naredbi koje promatrani sustav prihvaća i na koje može dati odgovor. U kontekstu pojedinih modula sustava postoji set naredbi koje modul prihvaća za interakciju s vanjskim sustavom ili korisnikom.

Učitavanjem dostupnih biblioteka u memoriju mikrokontrolera omogućili smo pozivanje određenoga seta naredbi specifičnih za specifični programski modul. *API Board_init*() podržava inicijalizaciju vremenske baze, vanjske *DDR* memorije, postavljanje *pinmuxa* i konfiguracije I/O sučelja. Biblioteke dodajemo standardnom naredbom C jezika u primjeru danom u nastavku:

```
# include <ti/board/board.h>
```

Nakon dodavanja biblioteke potrebno je izvršiti dodatne konfiguracije, nakon čega možemo koristiti komunikacijsko sučelje ili *API*.

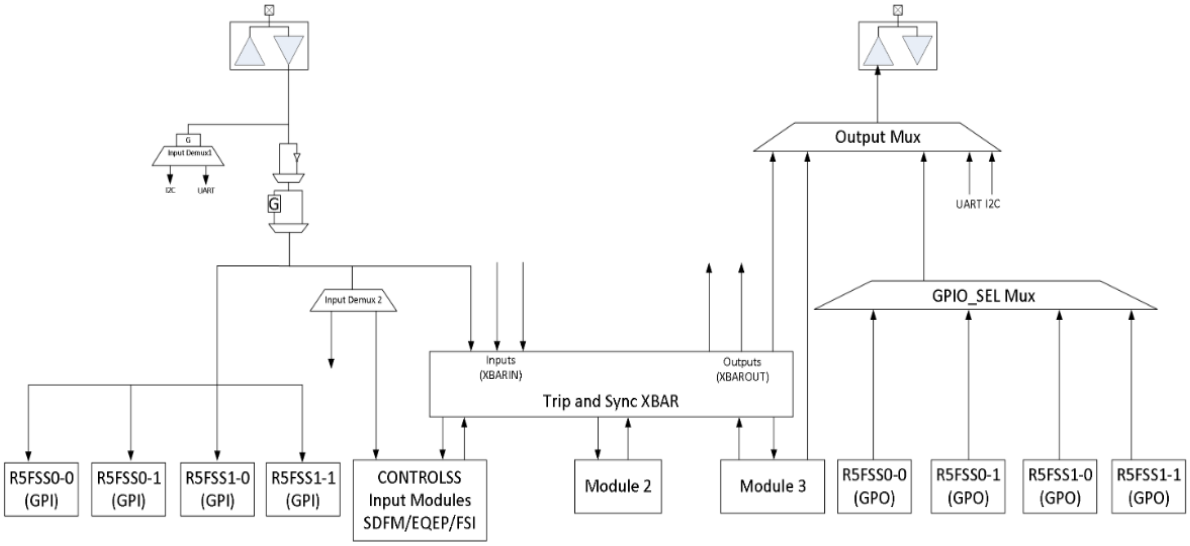
```
/* Set pinmux & UART */
Board_STATUS ret;
Board_initCfg boardCfg;

boardCfg = BOARD_INIT_MODULE_CLOCK | BOARD_INIT_PINMUX_CONFIG |
BOARD_INIT_UART_STDIO;

ret = Board_init(boardCfg);
```

Aplikacija treba konfigurirati postavke poput *BOARD_INIT_MODULE_CLOCK* kako bi *I2C* ili *Inter-Integrated circuiit* komunikacija radila. *I2C* se koristi za čitanje podataka iz *EEPROM* memorije. Funkcija koja brine za ponašanje *I2C* bit će kreirana u načinu ispitivanja ili *polling mode.* Ona traje do trenutka kada se identificirajuća oznaka ploče dobije iz *EEPROM* memorije putem *Board_getIDInfo() API* naredbe sučelja. *I2C* komunikacijski protokol se koristi za konfiguraciju I/O proširenja i multipleksora kako bi programski kod mogao pravilno funkcionirati na mikrokontroleru. Upravljačke funkcije za *I2C* bit će dostupne kada modul radi u *polling* načinu rada i zatvorene nakon što se inicijalizacija modula s multipleksorom dovrši. Potrebno je koristiti funkciju *Board_init()* s opcijom *BOARD_INIT_UART_STDIO* kako bi se koristio *UART API*. Nakon što *Board_init()* funkcija završi izvedbu, aplikacija može pozivati funkcije *UART*-a kao što su *UART_printf*, *UART_scanFmt* i slično.

Tako i za međuprocesnu komunikaciju također postoji *API*. Međuprocesna komunikacija ima pridružen *API* koji nije vezan za određeni procesor i može se koristiti za komunikaciju između procesora u višeprocesorskom okruženju (između jezgara), komunikaciju s drugim nitima na istom procesoru (između procesa) i komunikaciju s perifernim uređajima. Za stvaranje poštanskoga

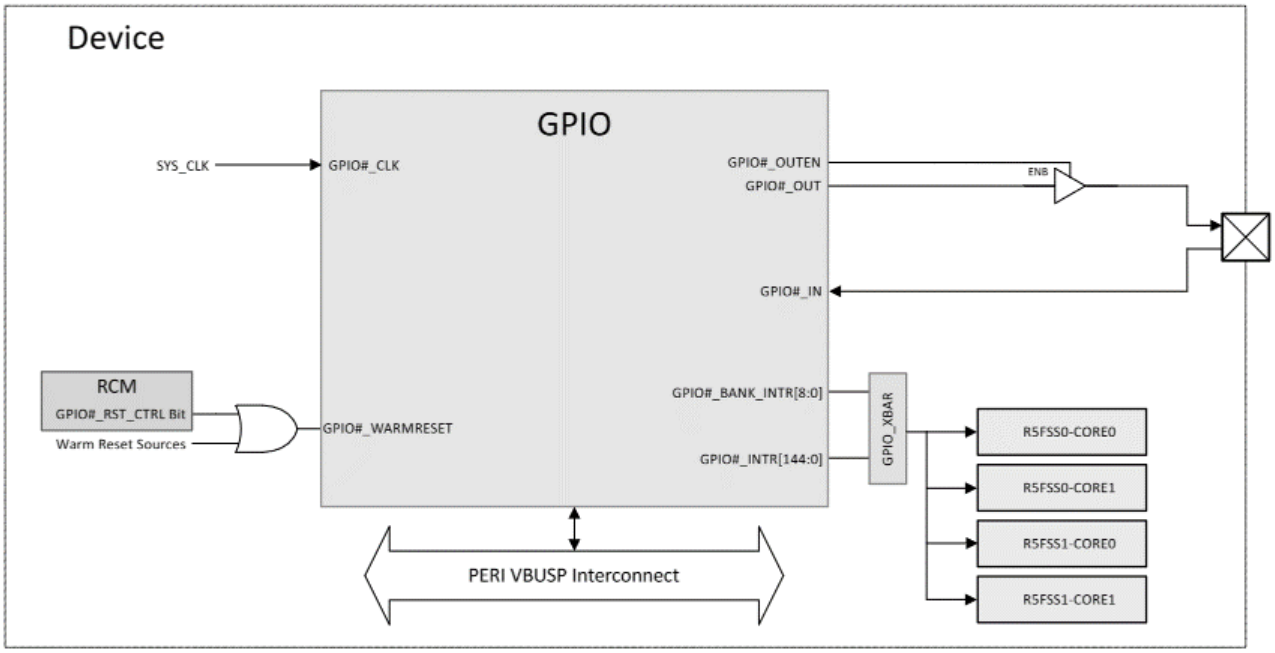
sandučića koji smo koristili prilikom međuprocesne komunikacije upotrijebljen je *API Mailbox_create*() koji je inicijaliziran sljedećim kodom:

*Mailbox_Handle Mailbox_create(SizeT bufsize, UInt numBufs, Mailbox_Params *params, Error_Block *eb)*



Slika 7. Shema GPIO te multipleksor i XBAR modula i njihova međusobna povezanost [31]

Svaka *R5FSS* jezgra mikrokontrolera ima pristup svim *GPIO* signalima. *GPIO* signali se mogu dodijeliti određenoj *R5FSS* jezgri konfiguriranjem *MSS_IOMUX.PAD_CFG_REG.GPIO_SEL* i povezanog *IOMUX Pad Configuration* registara. Sljedeći dijagram opisuje povezanost *GPIO* multipleksora.



Slika 8. GPIO shema [31]

DMA [32] ili *direct memory access* je računalna tehnika direktnoga pristupa memorijskim lokacijama. U danom kontekstu radi se o upravljačkoj jedinici unutar sustava čija je zadaća izvođenje upravo ove funkcionalnosti. Glavna stvar kod *DMA* upravljačke jedinice je da ona preuzima opterećenje s glavne procesorske jedinice kako bi se ona rasteretila. Tako vođenje računa o zahtjevima za pristup pojedinim memorijskim lokacijama spada upravo na *DMA* modul.
U *DMA-u* se koriste redovi kad god je potrebna komunikacija između entiteta.

Red podrazumijeva podatkovnu strukturu čija je svrha popis strogo uređenih unosa koji se obično koristi za prijenos informacija između proizvođača i potrošača. Generalno, to je apstrakcija koja reprezentira podatkovni niz na način da se novi unosi dodaju na kraj reda, dok se konzumiraju s početka reda. Unosi u redovima u većini slučajeva su reference na radno okruženje koje se prenosi, ali u nekim slučajevima (na primjer, paketi zahtjeva za prijenos) unosi u redovima mogu sadržavati stvarne podatke koji se prenose. Redovi mogu imati više različitih implementacija, a *DMA* koristi dvije najčešće: povezane liste i prstenove.

Povezane liste su podatkovna struktura koja implementira apstrakciju reda. Svaki unos pohranjuje ne samo podatke unosa već i pokazivač povezivanja na sljedeći unos na listi. Posljednji unos u

svakoj listi ima postavljen pokazivač na *NULL* vrijednost (obično kodiran kao 0x0). Upravitelj liste održava pokazivač na glavni element na listi i na posljednji element na listi. Budući da je pokazivač povezivanja pohranjen s podacima unosa, povezane liste imaju duljinu koja se dinamički mijenja i ograničena je samo mogućnošću dodjeljivanja dodatnih unosa koji će se postaviti u red za dodavanje/izdvajanje. Povezane liste prisutne su u opisima domaćina kako bi povezale više opisa i formirale paket.

Kružni međuspremnik je podatkovna struktura u kojoj se kontinuirani memorijski blok definiran s M, N-bajtovima (ukupna veličina je M × N bajtova) statički alocira i sekvencijalno piše/čita redom za prijenos podataka. Za implementaciju ove strukture također je moguće koristiti red kao i kod povezane liste, uz preinaku dodavanja dodatnoga pokazivača koji pokazuje na samu memorijsku lokaciju ove strukture.

5. INTERPROCESORSKA KOMUNIKACIJA (IPC)

Tablica 2. u dodatku predstavlja popis globalnih memorijskih lokacija. Na nju se možemo referirati za točne memorijske lokacije gdje se nalazi pojedini *SOC* (*System On a Chip*) moduli. Tako smo u slučaju *RPMessage* načina međuprocesne komunikacije mogli vidjeti na kojoj su memorijskoj lokaciji potrebni resursi za slanje poruka.

Kada je omogućena *IPC RP* poruka, zajednička memorija koristi se za razmjenu međuspremnika paketa između različitih procesora. Ova zajednička memorija, i ovdje je potrebno eksplicitno naglasiti, mora biti mapirana na istu adresu na svim procesorima jer u protivnome slanje poruka neće raditi, a vrlo je teško naknadno zaključiti gdje je točno pogreška. To se radi putem konfiguracijske datoteke, kao što je prikazano u donjem isječku preuzetom iz primjera *IPC RP* poruke, te korištenjem dijeljene memorijske lokacije na adresi koju smo dobili iz spomenute tablice u dodatku. [33]

```
(USER_SHM_MEM : ORIGIN = 0x701D0000, LENGTH = 0x00004000)
/* specify the memory segment */
MEMORY
{
...
USER_SHM_MEM : ORIGIN = 0x701D0000, LENGTH = 0x00004000
LOG_SHM_MEM : ORIGIN = 0x701D4000, LENGTH = 0x00004000
RTOS_NORTOS_IPC_SHM_MEM : ORIGIN = 0x701D8000, LENGTH = 0x00008000
}
/* map the shared memory section to the memory segment */
SECTION
{
...
/* General purpose user shared memory, used in some examples */
.bss.user_shared_mem (NOLOAD) : {} > USER_SHM_MEM
.bss.log_shared_mem (NOLOAD) : {} > LOG_SHM_MEM
/* this is used only when IPC RPMessage is enabled, else this is not used */
.bss.ipc_vring_mem (NOLOAD) : {} > RTOS_NORTOS_IPC_SHM_MEM
}
```

Za *IPC RP Message* potrebno je definirati samo dio memorije:

RTOS_NORTOS_IPC_SHM_MEM.

Međutim, može se konfigurirati segment zajedničke memorije za zajedničko zapisivanje određenih informacija za otklanjanje pogrešaka

LOG_SHM_MEM

te segment zajedničke memorije za generičku uporabu korisničkih aplikacija

USER_SHM_MEM.

Početna adresa za ove segmente može biti bilo koja, ali mora biti potpuno ista za sve procesore. Ostali segmenti memorije, osim zajedničkih memorija kao što su kodni ili podatkovni segmenti na svim procesorima, trebaju biti međusobno ne preklapajući. Inače će svaki procesor međusobno ometati i sustav neće raditi prema očekivanjima. Potrebno je ažurirati modul

MMU/MPU (Memory Management Unit i Memory Protection Unit)

za mikroprocesor s potrebnim parametrima. Zajednički segmenti memorije, koji se nalaze u datoteci s naredbama za povezivanje, moraju se mapirati kao *NON-CACHE* na procesorima s *RTOS/NORTOS* operativnim sustavom. Ovo se može postići putem *SysConfig* alata dodavanjem dodatnih unosa pomoću modula *MPU* .

Jezgre mikroprocesora moguće je međusobno sinkronizirati u izvođenju. Za to se može upotrijebiti metoda iz programskoga sučelja:

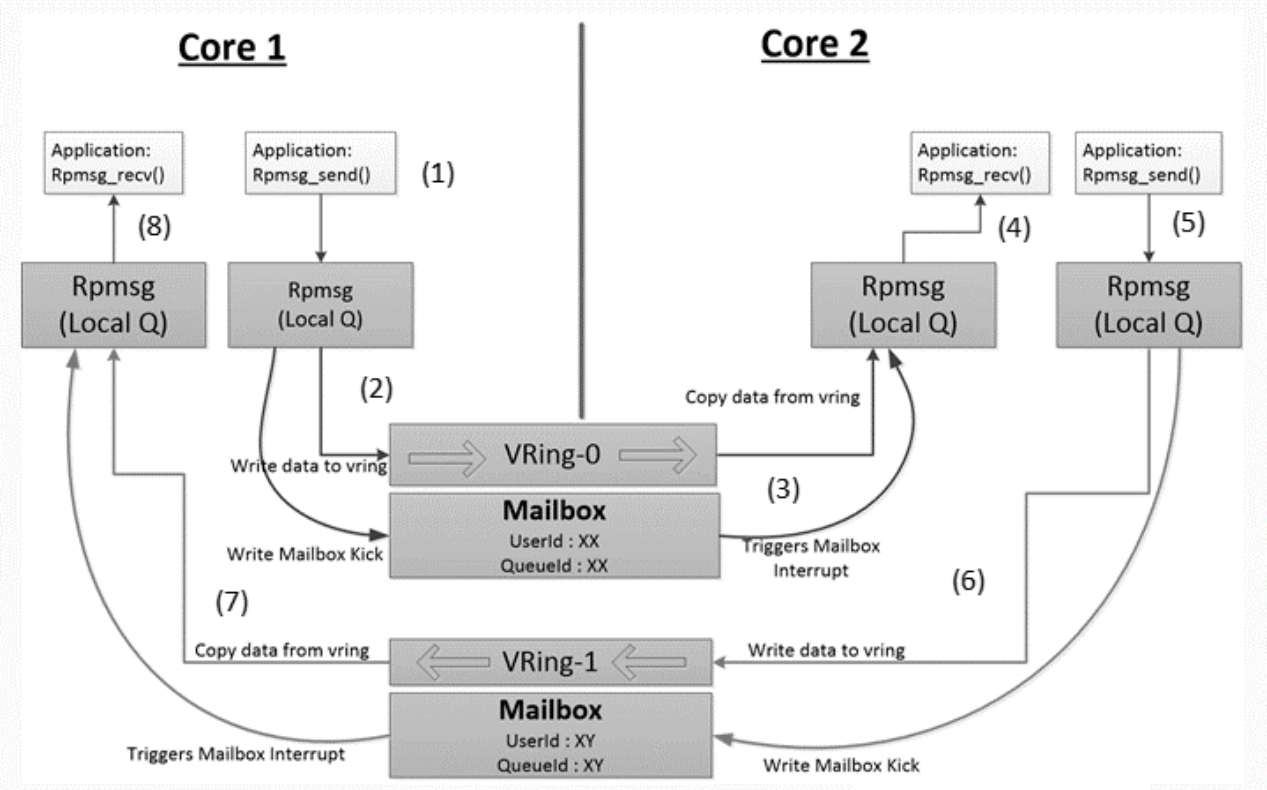
IpcNotify_syncAll(SystemP_WAIT_FOREVER);

Upotrebom ove naredbe nit izvođenja na jednom procesoru čeka signal s jedne ili preostalih jezgri da su ostali zadatci obavljeni i da se izvođenje može nastaviti. Ovo je korisno prilikom inicijalizacije kako bi osigurali pravilan redoslijed pokretanja pojedinih modula sustava. Tako je *TCP* poslužitelj obično sporiji prilikom inicijalizacije jer mora čekati dodjelu resursa od strane

mreže što su dugotrajne operacije, barem gledano naspram vremena inicijalizacije regulacijske strukture.
Nakon ovoga postupka konfiguracije moguće je slati poruke između jezgri procesora koristeći *API* sučelja definirana u *IPC Notify* i *IPC RPMessage* klasama.

Na komunikaciji među jezgrama mikroprocesora potrošeno je podosta vremena prilikom razvoja ovoga projekta. Razlog tomu je što projekti koji uključuju međuprocesnu komunikaciju zahtijevaju dodatnu konfiguraciju u *Sysconfig* alatu, pa i u ostalim dijelovima sustava. Nakon izrade inicijalnih dijelova projekata, *TCP* poslužitelj i regulator za motor, bilo je potrebno te sastavne dijelove projekta objediniti. To se postiže kreiranjem posebnoga tipa projekta – projekt kao sustavna datoteka.

Budući da imamo dva projekta, povezat ćemo ih pomoću trećega projekta koji je drukčijega tipa nego standardni projekti i služi upravo pružanju okvira ili ljuske kako bi sustav mogao povezati željene projekte. Jedan od projekata predstavlja dio sustava za komunikaciju, kako je opisano u prethodnim poglavljima, dok je drugi dio regulacijska struktura koja vrši generiranje upravljačkih signala čiji period aktivnoga stanja naspram ukupnoga perioda jednoga ciklusa korelira s postotkom udjela referentne vrijednosti naspram maksimalno moguće vrijednosti toga parametra te služi kao signal koji mijenja stanja na *IGBT* sklopkama u izmjenjivaču za propuštanje energije prema motoru.

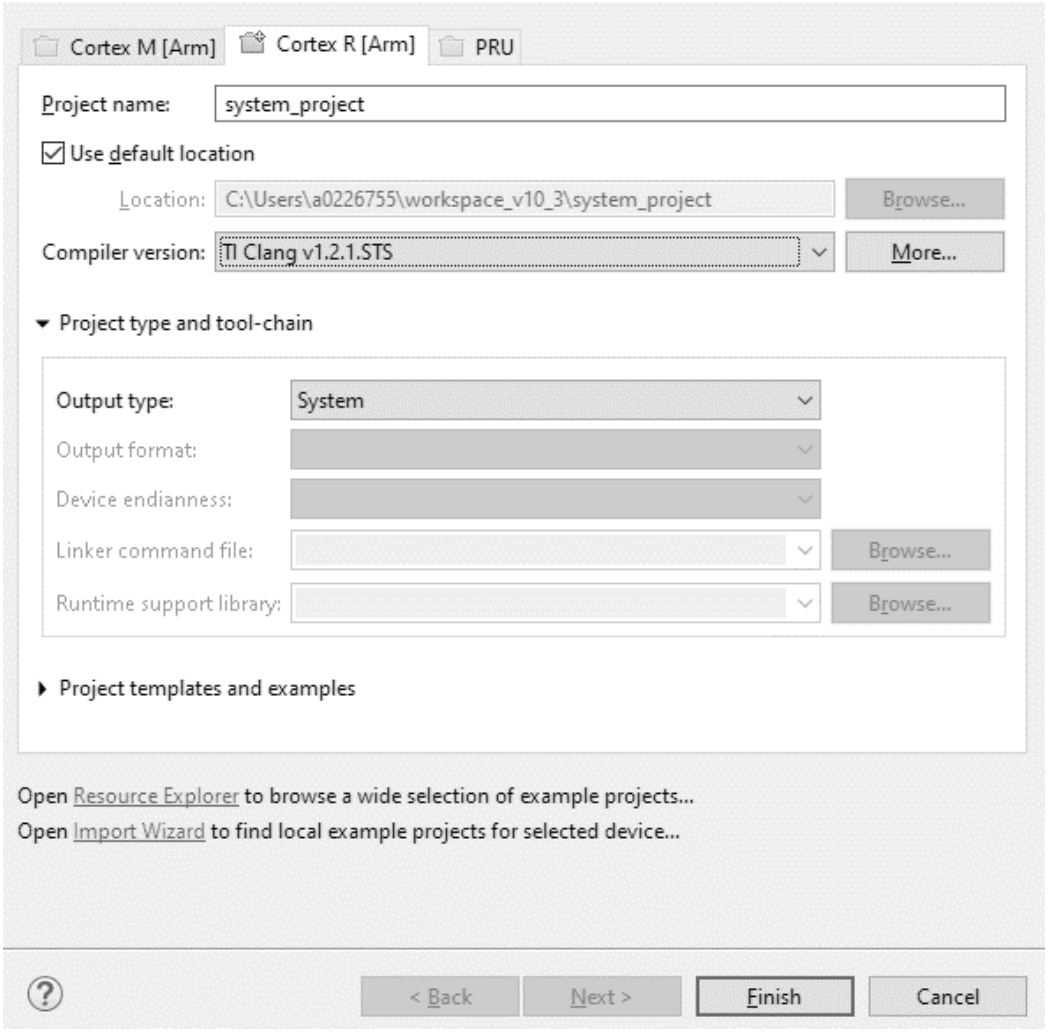


Slika 9. Funkcionalna shema IPC komunikacije [33]

Na Slici 9. vidimo osnovni princip korištene međuprocesne komunikacije. Počevši od točke 1, u aplikaciji zovemo *API* metodu

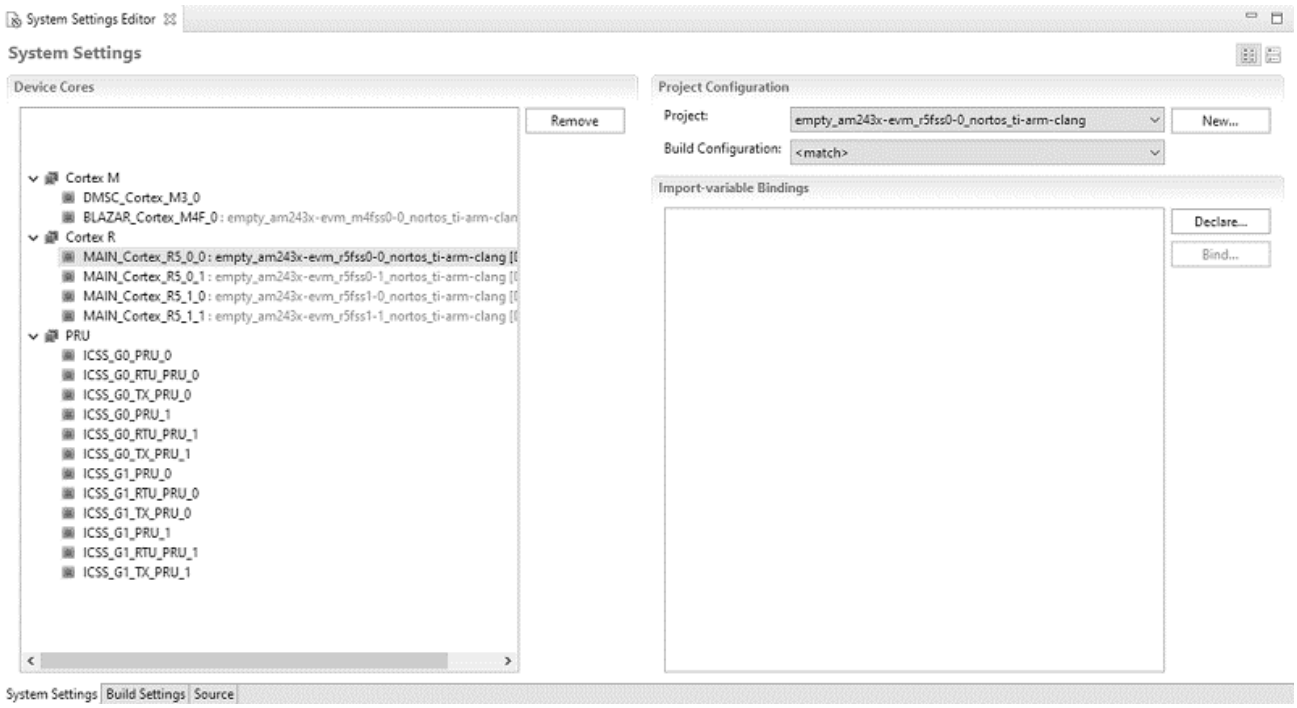
Rpmsg_send().

Kao parametar poslat ćemo konfiguracijske podatke poput odredišta, sadržaja poruke koji šaljemo i ostale. U tranziciji 2 poruka se prosljeđuje iz lokalnoga spremnika poruka u *VRing*. On predstavlja zajednički memorijski prostor u koji pristup imaju obje (ili više) jezgre. Upisivanjem vrijednosti u *VRing* odašilje se okidački signal prema drugoj jezgri u obliku prekida (*interrupt*). Taj prekid druga jezgra u tranziciji 3, aktiviranoj prekidom, obradi na način da kopira sadržaj iz *VRinga u* lokalni spremnik, ovaj put od primatelja poruke. Nakon toga ga čita iz lokalnoga spremnika primatelja. Isti proces vrijedi i za suprotan smjer slanja poruka.



Slika 10. Čarobnjak za izradu projekta

Kako bi se ostvarila interprocesorska komunikacija, bilo je potrebno pokrenuti čarobnjak za izradu projekta. Izgled sučelja za konfiguraciju prikazan je na gornjoj slici (Slika 10.) Zatim je potrebno dati ime projektu te u izborniku za *Output type* odabrati *System*. Otvara se novi prozor za konfiguraciju toga projekta (Slika 11.). U njemu konfiguriramo lokacije izvorišnih datoteka za izvršavanje pojedinoga projekta, kao i jezgru na kojoj će se kod za projekt izvoditi.



Slika 11. Konfiguracijski prozor za mapiranje projekta na jezgru

Postoje opcije i za mijenjanje izvornoga koda (.xml), no sva konfiguracija može se izvesti preko grafičkoga sučelja pa zbog jednostavnosti, modifikacije izvornoga koda ovoga projekta nećemo promatrati. Kad dodjeljujemo projekte na pripadajuće jezgre, potrebno je konfigurirati i memorijski prostor. Pošto sve jezgre koriste istu memoriju, potrebno je definirati, to jest raspodijeliti memorijske lokacije kako bi svaka jezgra imala svoj posebni privatni memorijski prostor, uz zajednički memorijski prostor u koji mogu zapisivati poruke koje želimo dijeliti između jezgri, kao što je *IPC* primjer u našemu slučaju. Tako u Tablici 3. u dodatku možemo vidjeti dostupni memorijski prostor te prikladno konfigurirati isti za izvođenje potrebnih funkcionalnosti sustava. Primjer koda za primanje RP poruke između procesorskih jezgri:

```
    int32_t status = RPMessage_recv(
        &gRecvMsgObject,    //(* deref.)primatelj poruke
        receveBuffer,       //međuspremnik za primanje podataka
        &replyMsgSize,      //(* deref.)velicina poruke
        &remoteCoreId,      //(* deref.)identifikator odredišne jezgre
        &remoteCoreEndPt,   //(* deref.)identifikator "porta" odredišne jezgre
        1);
```

Prikazane su informacije koje su zapisane u zabilješke o izvršavanju programa. Nakon inicijalizacije, server očekuje ulaznu vezu. Nakon zaprimanja dolazne veze, ostvarivanja procesa rukovanja s klijentom i čekanja na inicijalizaciju modula regulacije motora, server očekuje daljnje naredbe za upravljanje motorom. Kad je naredba za pokretanje motora zaprimljena od strane poslužitelja servisa, ispisuje se informacija o događaju te se šalje *IPC* poruka prema drugoj jezgri. Ona tu poruku – naredbu zaprima te obrađuje. Kako je riječ o naredbi za pokretanje motora, u regulacijskoj strukturi sprema se logička vrijednost koja označava stanje pokrenute ili zaustavljene simulacije. Kad regulator prođe jednu iteraciju upravljačke logike (*FOC*) te primijeti stanje varijable *ON/OFF* u logičkoj jedinici, započinje proces vrtnje motora, to jest simulacije iste. Pali se indikatorska *LED* dioda na pločici putem konfigurirane nožice za upravljanje stanje diode, dok se istovremeno pokreće proces proračuna potrebnih parametara regulatora motora, s obzirom na postavljene koeficijente unutar aplikacije.

```
=========================
  Ritroller ::  TCP SERVER
=========================
EnetPhy_bindDriver: PHY 3: OUI:080028 Model:0f Ver:01 <-> 'dp83869' : OK
PHY 3 is alive
PHY 12 is alive
Starting lwIP, local interface with static IP 192.168.1.10
Host MAC address-0 : 34:08:e1:77:fb:6f
[LWIPIF_LWIP] NETIF INIT SUCCESS
Enet IF UP Event. Local interface IP:192.168.1.10
[LWIPIF_LWIP] Enet has been started successfully
Waiting for network UP ...
Waiting for network UP ...
Cpsw_handleLinkUp: Port 1: Link up: 1-Gbps Full-Duplex
MAC Port 1: link up
Network Link UP Event
Network is UP ...
Waiting on Motor Control module to init ...
Motor Control module init complete!
.[?] Awaiting new connection...
.[!] Accepted new connection... (70118E40)
..[!]Received command b:1 --> Starting Motor..
CPU load =   1.46 %
..[!]Received command b:0 --> Stopping Motor..
CPU load =   1.50 %
..[!]Received command b:1 --> Starting Motor..
CPU load =   1.49 %
..[!]Received command b:0 --> Stopping Motor..
CPU load =   1.50 %
..[!]Received command b:1 --> Starting Motor..
CPU load =   1.50 %
```

Slika 12. Prikaz zapisnika naredbi prilikom zaprimanja naredbi za pokretanje i zaustavljanje motora

Simulacija uključuje *Tick* komponentu u strukturi koja predstavlja vrijeme, točnije diskretizirani trenutak u izrazu:

$$x[n] = x_a\,(nT) \quad (5.1)$$

gdje *Tick* koji nalazimo u programskom smislu predstavlja varijablu „n“ u gornjem izrazu dok je T period uzorkovanja. *DcBus* varijabla na donjoj slici (Slika 14.) predstavlja istosmjerni napon napajanja. Vidljive su i Ia, Ib te Ic komponente struje koje koristimo u *Clarke* i *Park* transformacijama za pretvorbu iz abc u $\alpha\beta$ pa zatim u dq sustave. Za te transformacije potrebni su i električni i mehanički kutovi te izlazne vrijednosti napona za d i q os, pošto na integrirane PI regulatore dovodimo konstantni signal pogreške. Također su prikazane i izlazne vrijednosti upravljačkoga signala pojedinoga regulatora, implementiranoga u digitalnome obliku.

Expression	Type	Value
packet	struct InverterStreamPacket_t	{...}
IsrTick	unsigned int	2224540
Ia	float	1.36010051
Ib	float	1.51388395
Ic	float	0.00756311696
DcBus	float	225.076309
Id	float	-0.957078516
Iq	float	0.00165376067
theta_e	float	1.14182758
omega_e	float	-0.00462193927
Out_Vd	float	76.4019165
Out_Vq	float	-26.8954983
RegSpeed_Fback	float	0.0
RegSpeed_Output	float	0.0
RegId_Fback	float	-0.957078516
RegId_Output	float	100.0
RegIq_Fback	float	0.00165376067

Slika 14. Prikaz podatkovne strukture paketa poruke za IPC

Na Slici 15. je prikazan zapisnik iz izvođenja prilikom slanja naredbe za dohvaćanje podataka s mikrokontrolera. Nakon uspostavljanja veze te pokretanja motora, upisuju se podatci poslani putem *TCP* veze prema komunikacijskom servisu zaduženom za razmjenu podataka između sučelja i mikrokontrolera.

```
=========================
  Ritroller ::  TCP SERVER
=========================
EnetPhy_bindDriver: PHY 3: OUI:080028 Model:0f Ver:01 <-> 'dp83869' : OK
PHY 3 is alive
PHY 12 is alive
Starting lwIP, local interface with static IP 192.168.1.10
Host MAC address-0 : 34:08:e1:77:fb:6f
[LWIPIF_LWIP] NETIF INIT SUCCESS
Enet IF UP Event. Local interface IP:192.168.1.10
[LWIPIF_LWIP] Enet has been started successfully
Waiting for network UP ...
Waiting for network UP ...
Cpsw_handleLinkUp: Port 1: Link up: 1-Gbps Full-Duplex
MAC Port 1: link up
Network Link UP Event
Network is UP ...
Waiting on Motor Control module to init ...
Motor Control module init complete!
.[?] Awaiting new connection...
.[!] Accepted new connection... (7011A440)
..[!]Received command b:1 --> Starting Motor..
..[!]From Motor(ret): data: 22520,1.544136,1.455900,-0.119749,225.041748,-1.081028,-0.006237,0.994287,0.063008,73.502
823,-16.778107,0.000000,0.000000,-1.081028,100.000000,-0.006237,92.246307,0.000000,0.000000,1.000000,0,0.000000,0.001
000,0.000800,0.000000,1.000000,0.100000,0.000000,1.000000,0.100000


..[!]From Motor(stringMessage): data: 22520,1.544136,1.455900,-0.119749,225.041748,-1.081028,-0.006237,0.994287,0.063
008,73.502823,-16.778107,0.000000,0.000000,-1.081028,100.000000,-0.006237,92.246307,0.000000,0.000000,1.000000,0,0.00
0000,0.001000,0.000800,0.000000,1.000000,0.100000,0.000000,1.000000,0.100000


CPU load =   1.48 %
```

Slika 15. Konzolni ispis vrijednosti stanja motora

Struktura podataka ovdje nije vidljiva, već samo vrijednosti poslanih parametara kako se ne bi zagušio pregled u konzoli s previše podataka i kako bi se postigla urednost i preglednost. Mikrokontroler šalje podatke u intervalima od 100 milisekundi, kako je i namješten interval osvježenja grafičkoga prikaza ovih podataka na korisničkom sučelju. Ovi intervali su sinkronizirani kako ne bi došlo do odašiljanja neprikladnoga broja podataka.

Na primjer, ukoliko šaljemo paket svakih 100 milisekundi, a čitamo ih svakih 300 milisekundi, u tom intervalu dva paketa će biti poslana prema odredištu dok će odredište pročitati samo jedan paket ili će se prepuniti međuspremnik, ovisno o konfiguraciji i načinu čitanja podataka iz istoga. Isto vrijedi ukoliko čitamo svakih 100 milisekundi, a šaljemo svakih 300 milisekundi, ponovo dolazi do nepotrebnoga korištenja resursa što dovodi do nekonzistentnosti načina izvršavanja i može prije ili kasnije dovesti do greške. No prilikom pravilne konfiguracije pročitani paketi na odredišnoj strani čitaju se te se putem *TCP* protokola šalju na poslužitelj. Poslužitelj podatke raščlanjuje na pojedine parametre, pri čemu dobijemo mogućnost individualnoga promatranja pojedinoga parametra. Iste zbog raščlanjivanja možemo spremiti u bazu podataka poštujući 1. normalnu formu te je prikladne podatke lakše proslijediti na potreban prozor ili polje unutar prikaza na korisničkom sučelju.

6. KORISNIČKO SUČELJE

Ova komponenta predstavlja korisničko sučelje aplikacije koje omogućuje korisnicima interakciju sa sustavom. *GUI (engl.* ”*Graphical User Interface”*) djeluje kao glavna točka interakcije između korisnika i sustava, omogućavajući korisnicima učinkovito praćenje i upravljanje povezanim *hardverom*. U ovome dijelu bit će prikazana arhitektura, funkcionalnost, princip te čemu je pridodana pažnja prilikom izrade.

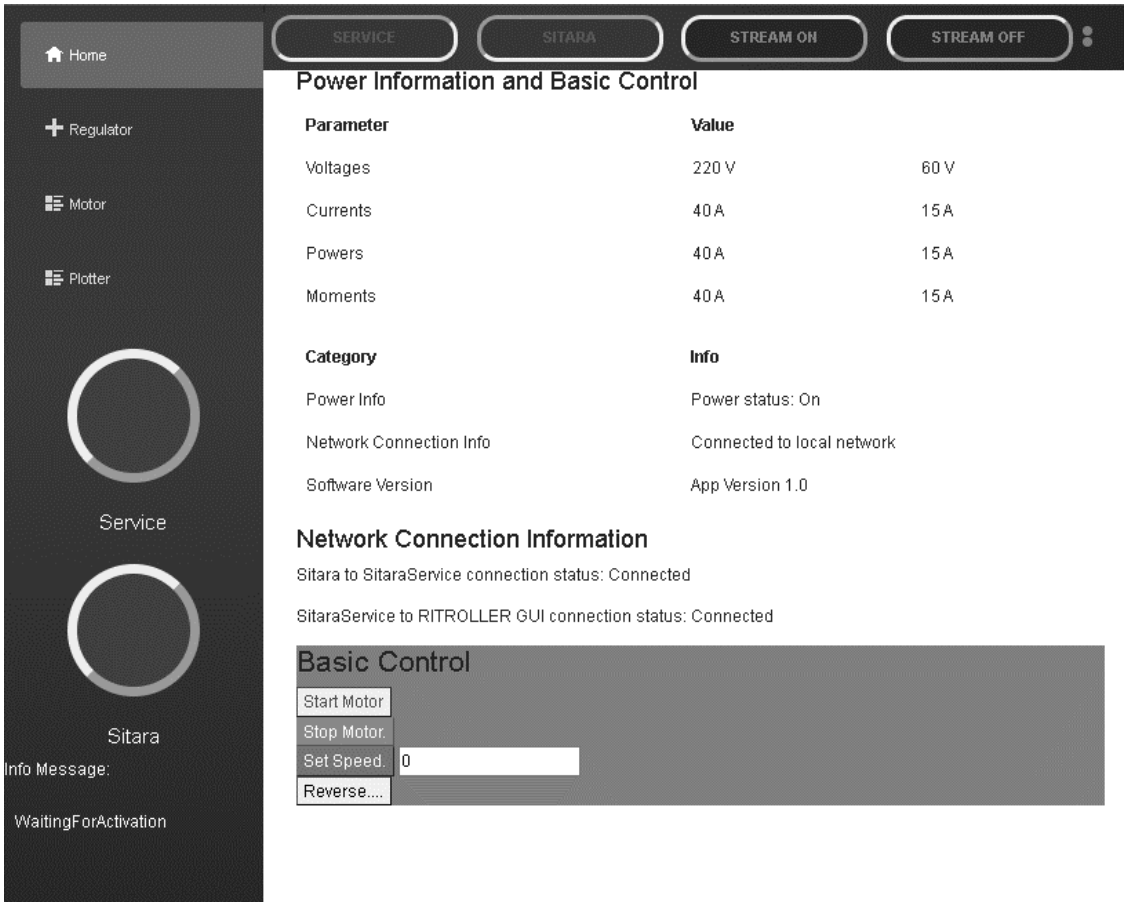
GUI je izgrađen kao mrežna aplikacija, što ga čini dostupnim putem standardnih mrežnih preglednika na raznim uređajima. Ovaj izbor donosi brojne prednosti, uključujući kompatibilnost s različitim platformama i mogućnost daljinskoga pristupa. *GUI* se izvodi na zasebnom poslužitelju, omogućavajući korisnicima interakciju sa sustavom putem mreže. Korisnik se jednostavno s bilo kojim internetskim preglednikom može spojiti na lokalnu adresu na kojoj je pokrenut program (ovo je proširivo i na vanjske adrese) i upravljati sustavom. Ovaj model klijent - poslužitelj omogućava fleksibilnost u implementaciji i osigurava besprijekornu komunikaciju s ugrađenim komponentama.

Značajan izazov bio je pružanje vizualizacije stvarnih podataka u stvarnome vremenu. *GUI* je opremljen za prikaz stvarnih senzorskih podataka, metrika performansi motora i statusa sustava. *GUI* se mora prilagoditi različitim veličinama zaslona, od malih mobilnih ekrana do velikih radnih monitora. Korišten je *CSS* i principi prilagodljivoga dizajna stvarajući raspored elemenata koji se prilagođava različitim veličinama zaslona. Korisničko sučelje dizajnirano je kako bi bilo praktično za korištenje. Sadrži intuitivan raspored s nadzornom pločom na kojoj se prikazuju najvažnije informacije. Učinkovit prijenos podataka između korisničkoga sučelja i ugrađenoga sustava ključan je za stvarno vrijeme ažuriranja. Koristili smo asinkrone komunikacijske protokole poput *WebSocket* protokola kako bismo osigurali minimalno kašnjenje. *WebSocket* omogućuje dvosmjernu komunikaciju, omogućavajući sustavu slanje ažuriranja prema korisničkom sučelju čim postanu dostupna.

Tijekom razvojnoga procesa bilo je nužno suočiti se s raznovrsnim izazovima. Jedan takav izazov bio je održavanje sinkronizacije između korisničkoga sučelja i poslužitelja. Ovaj izazov riješen je implementacijom arhitekture temeljene na događajima ili engleski *events.* Sustav šalje informacije

o promjenama na korisničkom sučelju za *event* na koji smo se pretplatili. Taj pristup smanjuje potrebu za kontinuiranim ispitivanjem mikrokontrolera i optimizira ukupne performanse cijeloga sustava.

Drugi izazov bio je optimizacija vizualizacije podataka. Brza i precizna vizualizacija senzorskih podataka zahtijeva značajne računalne resurse sa svrhom osiguranja izvedbe bez zastoja.
Početna stranica sadrži osnovne informacije o sustavu. Na vrhu stranice nalaze se tipke koje izvršavaju pripadajuće naredbe, to su pokreni i zaustavi motor, postavi željenu brzinu te reverziraj motor. Svaka naredba preko do sada opisane arhitekture naredbu prosljeđuje do mikrokontrolera na kojoj pomoću međuprocesorske komunikacije prosljeđujemo naredbu regulacijskoj strukturi te se ona izvrši.



Slika 16. Početni zaslon aplikacije

Pri vrhu prozora aplikacije nalazi se komandna traka. U toj traci nalaze se gumbi na čiji pritisak se pokreću glavne operacije za interakciju s mikrokontrolerom. Tako ovdje imamo gumb za spajanje sa servisom. Iako se spajanje na servis može izvršiti automatski, bez potrebe korisnika za interakcijom, ovdje je namjerno implementiran eksplicitni zahtjev da korisnik mora pritisnuti gumb za spajanje s poslužiteljem kako bi se demonstrirala sposobnost korištene tehnologije. Tako postoji i gumb za spajanje, to jest kreiranje *TCP* veze s mikrokontrolerom.

Šalje se testni paket prema mikrokontroleru te se u *TCP* klasi koju koristimo za uspostavljanje veze gleda status veze, to jest ako je veza uspješno uspostavljena. Ukoliko je, korisnika se obavještava o stanju sustava i o stanju ostvarenih veza između entiteta sustava tako da gumb mijenja boju sukladno stanju veze. Dodatno, ukoliko postoji pogreška u ostvarivanju veze na nekom dijelu sustava, informacija o pogreški zbog koje se veza ne može ostvariti ispisuje se u polje namijenjenom za prikaz pogrešaka, koje se nalazi ispod statusnih indikatora.

Ispod polja za prikaz električnih vrijednosti koje promatramo na mikrokontroleru nalazi se prozor s osnovnim naredbama za upravljanjem motorom. Gumbi su povezani na ostatak sustava pozadinskim kodom korisničke aplikacije. Pokreću niz asinkronih poziva funkcija koje čine ranije opisane međuslojeve aplikacije. Tako događaj pritiska pojedinoga gumba poziva istu metodu, pošalji naredbu, samo s različitim parametrima, ovisno o akciji koju želimo poduzeti. Na klik gumba pokretanje motora zove se metoda s parametrom *b:1*. Postavljanje brzine, s druge strane, uz parametar *b:3 - 200* šalje i prošireni dio parametra (200), s vrijednošću koji želi postaviti, u ovome slučaju brzinu vrtnje od 200 okretaja u minuti. Ovu metodu koristi i dio sustava za spremanje parametra regulatora i motora.

7. MODEL REGULACIJSKE STRUKTURE I STROJA

U ovome poglavlju bit će prikazana implementacija programskih struktura za modeliranje regulacijske strukture i stroja. Promatrano je definiranje parametara i kreacija potrebnih i povezanih metoda za izvršavanje funkcije pojedine komponente sustava. Regulacijski koncepti preuzeti su iz prijašnjega projekta gdje je regulacijska struktura implementirana u diskretnome obliku, uz razliku što je u tome projektu korišten izmjenični stroj kao aktuator. Iako postoje sličnosti što se tiče funkcionalnosti i parametara, diskretni i kontinuirani regulatori koriste fundamentalno drukčije principe na koji način generiraju upravljački signal s obzirom na ulazne vrijednosti.

7.1. Regulatori

Podatkovna struktura regulatora je kreirana u dijelu koda koji se odnosi na cijeli projekt i dostupan je svim podsustavima i slojevima aplikacije na korištenje. Bilo koja druga klasa ili objekt koji treba pristup strukturi regulatora ili pripadnim funkcionalnostima i metodama te klase može to učiniti s jednoga mjesta. Zadane su inicijalne vrijednosti prilikom kreiranja novoga objekta strukture *PI_OBJ*. Definirane su varijable za pohranu parametara regulatora. Iz primjera definicijskoga koda ove strukture vidljivi su korišteni parametri *PI* regulatora.

Tip podataka *float32* koristi se u programiranju za opisivanje brojeva s pomičnim zarezom jednostruke preciznosti. U skladu s *IEEE 754* [34] standardom ovaj tip podataka uobičajeno zauzima 32 bita ili 4 bajta memorije. Bitovi *float32* tipa podataka podijeljeni su na tri dijela, predznak, eksponent i mantis. Bit predznaka određuje hoće li broj biti pozitivan ili negativan. Tip *float32* daje ravnotežu između preciznosti i učinkovite upotrebe memorije. Ima manju preciznost od *float64*, ali dovoljnu za korištenje u ovome slučaju.

```
typedef struct _PI_Obj_
{
    float32_t Kp;              //!< the proportional gain for the
PI controller
    float32_t Ki;              //!< the integral gain for the PI
controller
    float32_t Ui;              //!< the integrator start value
for the PI
                               //!< controller
    float32_t refValue;        //!< the reference input value
    float32_t fbackValue;      //!< the feedback input value
    float32_t ffwdValue;       //!< the feedforward input value
    float32_t error;
    float32_t up;
    float32_t outValue;
    float32_t outMin;          //!< minimum output value for
controller
    float32_t outMax;          //!< maximum output value for
controller
    uint16_t integralLock;     //!< Lock integral gain for anti-
windup
} PI_Obj;
```

Deklarirane varijable pohranjuju vrijednosti za proporcionalno i integralno pojačanje, referentnu ili ulaznu vrijednost regulatora, signal greške, iznose krajnjih apsolutnih maksimalnih vrijednost te izlazni upravljački signal.

U metodi za regulator implementiran je *anti-windup* model. Taj model odgovoran je za eliminiranje nakupljanja regulacijske pogreške koja se pojavljuje od integracijske komponente regulatora u kombinaciji s ograničavanjem izlazne veličine. [35]

Windup fenomen se ostvaruje kada je razlika između referentne i mjerene vrijednosti velika po apsolutnom iznosu. U tom slučaju pogreška koja predstavlja spomenutu razliku vrijednosti je velika te se ona propagira prema regulatoru gdje na integracijskom bloku počinjemo sumirati veliku pogrešku koja će se akumulirati u bloku za kašnjenje, pri digitalnoj/diskretnoj implementaciji regulatora.

Izlazni regulacijski signal bit će ograničen maksimalnom dopuštenom vrijednošću limiterom na izlazu te na taj način da na stroj ne dovodimo više od dopuštenoga nazivnog iznosa reguliranih vrijednosti. Tako regulacijski sustav može ući u zasićenje, to jest ne može reagirati na povećanje ulazne pogreške dodatnim pojačanjem izlazne vrijednosti, dok se na integralnoj komponenti neprestano dodaje regulacijska pogreška. Tek kada se nakon nekoga vremena postigne postavljena

referentna vrijednost, pogreška je i dalje akumulirana u integralnoj komponenti regulatora te mora proći neko vrijeme da bi se regulator vratio u ispravno stanje gdje ne postoji pogrešno akumulirana pogreška.

Za potrebe sustava stvorena su dva regulacijska objekta. To podrazumijeva regulacijsku strukturu za d i q os sustava nakon Clarke i Park transformacija. Regulatori se pozivaju kao metode tijekom izvođenja upravljačke petlje te im se svi potrebni ulazni parametri šalju kao argument metode ili funkcije. Nakon proračuna metoda unutar klase koja reprezentira funkcionalnost *PI* regulatora u matematičkom smislu, metoda pozivatelju vraća proračunatu vrijednost za poslane parametre. Tada se vrijednost za d i q os dalje šalju prema dijelu sustava s funkcijama za proračun *PWM* signala za upravljačke sklopke.

Prilikom spremanja parametara pojedine komponente sustava upravljanja, bili to regulatori, tiristori ili sam električni stroj uz zapise u memoriju mikrokontrolera (koji nema trajnu pohranu podataka), oslanjamo se na bazu podataka koja je na raspolaganju sa strane poslužitelja. Bez iste, podatci i varijable za rad sa sustavom prisutne su isključivo prilikom izvođenja te nakon gašenja sustava, gube se sve informacije i podatci. U nastavku se nalazi programski kod u *C#* za spajanje na bazu podataka.
Korištena metoda za spajanje na bazu podataka prikazana je u nastavku.

```
private readonly AppDbContext _dbContext;

        public SitaraService(AppDbContext dbContext)
        {
            _dbContext = dbContext;
        }

        // Metoda za dodavanje podataka u bazu
        public async Task DodajPodatkeUBazu(string podaci)
        {
            var motorPodaci = new MotorPodaci { Podaci = podaci
};

            _dbContext.MotorPodaci.Add(motorPodaci);
            await _dbContext.SaveChangesAsync();
        }

        // Metoda za dohvaćanje podataka iz baze
        public List<MotorPodaci> DohvatiPodatkeIzBaze()
        {
            return _dbContext.MotorPodaci.ToList();
        }
```

Struktura podataka za bilježenje vrijednosti pozicije motora. Specifičan primjer koristi se kod *encoder* modula:

```
typedef struct
{
    /* Output: Motor electrical angle (Q15) */
    float thetaElec;
    /* Output: Motor mechanical angle (Q15) */
    float thetaMech;
    /* Output: Motor rotation direction (Q0) */
    int16_t directionQEP;
    /* Variable: Raw angle from timer 2 (Q0) */
    int16_t thetaRaw;
    /* Parameter: 0.9999 / total count, total count = 4000 (Q26) */
    float mechScaler;
    /* Parameter: Number of pole pairs (Q0) */
    int16_t polePairs;
    int16_t calAngle;
    uint32_t speedScaler;
    float speedPR;
    /* Skaliranje razlike kuta u brzinu */
    uint32_t K1;
    /* Skaliranje razlike kuta u brzinu */
    float K2;
    /* Skaliranje razlike kuta u brzinu */
    float K3;
    /* Output: Speed in rpm (Q0) - independently with global Q */
    float speedRPMPR;
    float oldPos;
    float speedFR;
    float speedRPMFR;
} PosSpeed_Object;
```

Sličan princip, samo s drukčijim nazivima, korišten je za spremanje ostalih parametara. Opet koristimo injektiranje te u spremnik servisa dodajemo *dbContext* klasu koju možemo koristiti za manipulacije nad bazama s pripadnim metodama. U metodi za dohvaćanje podataka iz baze potrebno je definirati tip podataka koje vraćamo, a u ovome slučaju dohvaćamo cijelu listu spremljenih motora, tipa *MotorData* pohranjenih u bazu. Naravno, može se dohvatiti i samo jedan motor konfiguriranjem parametara ili smanjivanjem inicijalne liste po željenom kriteriju, poput vremena unosa. Tako možemo pristupiti posljednjem unesenom motoru u sustav.

7.2. Električni stroj (asinkroni)

Jedna od glavnih funkcionalnosti aplikacije je sposobnost upravljanja strojem i spremanje varijabli tipa naponi i struje za daljnje korištenje. Izvorni parametri uzimaju se iz regulacijske strukture. Na temelju toga modela napravljeni su modeli na svim razinama aplikacije kako bi se postigla istovjetnost strukture koju koristimo. Pošto su mikrokontroler i poslužitelj dva odvojena sustava s različitom strojnom opremom koju koriste, pa time i procesorskom jedinicom koju koriste, svaki dio sustava mora posjedovati svoje definicije modela.

Unatoč tomu, one su istovjetne te posjeduju isti tip i broj varijabli koje opisuju pojedinu strukturu u oba dijela sustava. Kao što je opisano u dijelu rada vezanim uz komunikaciju dijelova podsustava, nakon pojedine interakcije između podsustavima, to jest upotrebe nekih komunikacijskih protokola, korak prije i nakon slanja informacija uvijek se svodi na pripremu podataka za slanje ili čitanje. To podrazumijeva pretvaranje iz binarnih u *ASCII* znakovlje, a zatim u niz smislenih rečenica ili naredbi/informacijskih paketa koje naposljetku interpretiramo s pojedine strane komunikacijskoga kanala.

Struktura podataka za bilježenje parametara motora dana je u nastavku:

```
typedef struct _Motor_t_
{
    float32_t  Ls_d;
    float32_t  Ls_q;
    float32_t  Rs;
    float32_t  Phi_e;
    float32_t  I_abc_A[3];          //!< the current values
    float32_t  V_abc_V[3];          //!< the voltage values
    float32_t  I_scale;
    float32_t  V_scale;
    float32_t  dcBus_V;             //!< the dcBus value
    float32_t  oneOverDcBus_invV;   //!< the DC Bus inverse, 1/V
    float32_t  I_ab_A[2];           //!< the current values
    float32_t  I_dq_A[2];           //!< the current values
    float32_t  theta_e;             //!< the rotor position
    float32_t  omega_e;             //!< the current values
    float32_t  Sine;
    float32_t  Cosine;
    float32_t  theta_e_out;
    float32_t  Sine_out;
    float32_t  Cosine_out;
    float32_t  Vff_dq_V[2];         //!< PI_dq FF value for decoupling
    float32_t  Vout_dq_V[2];        //!< the current values
    float32_t  Vout_ab_V[2];        //!< the current values
    float32_t  Vout_dq_out_norm;
    float32_t  Vout_dq_sat_gain;
    float32_t  Vff_max;
    float32_t  Vff_min;
    PI_Obj  pi_spd;
    PI_Obj  pi_id;
    PI_Obj  pi_iq;
    float32_t  vqLimit;
    float32_t  modulationLimitSquare;
    float32_t  Vout_max;
    float32_t  sampleTime;
    float32_t  outputTimeCompDelay;
    float32_t  resolverCompDelay;
    uint16_t  resolverCompIdx;
    uint16_t  isrResRatio;
} Motor_t;
```

Na stranici aplikacije za konfiguraciju parametara motora omogućili smo unos parametara pomoću polja za unos. Svako polje odgovara određenom parametru motora, poput referentne brzine vrtnje, otpora, induktiviteta za pojedinu os u kojoj promatramo motor (nakon transformacija iz abc u $\alpha\beta$ te u dq sustav). U nastavku je dan primjer koda za unos parametara s korisničkoga sučelja:

```
  <input type="number" @bind="motorValues[property.Name]" />
  <button @onclick="() => SendParameterAsync(property.Name,
motorValues[property.Name])"> Send </button>
```

Korištenje *@bind* parametra u *input* bloku nam omogućava praćenje promjena u *TextBox* polju unesenih od strane korisnika. Zatim te promjene šaljemo pripadnim metodama za daljnje manipuliranje podatcima. Atributi se odmah prosljeđuju prema mikrokontroleru kako bi nove parametre sustav mogao implementirati u regulacijsku strukturu. Korišteni su potpuno isti parametri tijekom razvoja aplikacije, što se tiče motora i regulatora. Kao i do sada, korisničko sučelje javlja poslužitelju da korisnik želi promijeniti parametar, šalje se paket na mrežu koji usmjerivač prosljeđuje do poslužitelja, a zatim do mikrokontrolera.

Poslužitelj zatim čita naredbu s korisničkoga sučelja, odlučuje koju metodu treba pozvati da bi obradio zahtjev klijenta, u ovome slučaju promjenu parametra. Sve metode su asinkrone kako bi se čekanje svelo na minimum. Time omogućujemo servisu za komunikaciju na poslužitelju da se ne zadržava na pojedinome zadatku, već da obrađuje nove zahtjeve. To mu daje mogućnost nastavka s radom na zadatcima koje bismo inače čekali. Nakon završetka dugotrajnih zadataka na koje čekamo, kontekst izvođenja ponovno se vraća na izvršavanje zadataka čiju smo operaciju čekali. Tako komunikacijski servis na poslužitelju može poslati parametar prema mikrokontroleru te u čekanju na njegovo izvršavanje može obavljati ostale zadatke, poput slanja podataka prema klijentu.

Nova upravljačka veličina stroja ulazi u proces obrade na jezgri mikrokontrolera zaduženoga za regulaciju stroja. Tijekom rada spremamo novu vrijednost te se regulacijski algoritam automatski prilagođava novoj zadanoj vrijednosti izvršavajući s njom regulacijski algoritam. Ponovni izračun se događa u svakoj iteraciji algoritma, kako to i je običaj u digitalnoj obradi signala i sustavima za rad u stvarnome vremenu.

Koristimo rječnik (*engl. Dictionary)* kao tip parametra [36] jer nam omogućava spremanje proizvoljnoga broja vrijednosti na jednostavan način, kako je to opisano u poglavlju „Komunikacijski servis“ [37]. Korištenjem rječnika možemo dodavati i uklanjati parametre dinamički prema potrebama. To znači da nismo ograničeni fiksnim brojem parametara, što je korisno kada radimo s motorima koji imaju različite konfiguracije. Rječnik omogućava povezivanje s poljima za unos na grafičkome dijelu aplikacije. Svako polje za prikaz na korisničkoj

strani povezano je s određenim ključem u rječniku, što olakšava unos i praćenje vrijednosti parametara te brz pristup i promjenu tih parametara u kodu.

Na primjer, kada trebamo promijeniti brzinu vrtnje motora, jednostavno tražimo ključ ili ime parametra *brzina_vrtnje* u rječniku i mijenjamo mu vrijednost. Korištenjem rječnika lako možemo pratiti promjene u parametrima. Kada se promijeni vrijednost, ona se automatski ažurira u rječniku. Ovdje ih možemo i pohraniti u bazu podataka što nam omogućava praćenje povijesti promjena parametara za buduće analize.

Kako strojevi mogu imati različite parametre, rječnik omogućava jednostavno dodavanje ili uklanjanje parametara bez velikih promjena u kodu. Možemo implementirati optimizacije kako bismo smanjili potrošnju memorije i poboljšali brzinu pristupa podatcima, na primjer, koristeći tipove podataka koji su optimalni za čuvanje vrijednosti parametara. Kada su parametri spremljeni u rječnik, šaljemo cijeli rječnik prema poslužitelju koji dalje komunicira s mikrokontrolerom i daje mu do znanja koje parametre je potrebno izmijeniti.

Kako bismo omogućili spremanje proizvoljnoga broja parametara u rječnik, dinamički dodajemo ključeve i vrijednosti prema unosu korisnika na korisničkome sučelju. To znači da se novi parametri pojavljuju u rječniku čim ih korisnik unese, a isto tako uklanjamo parametre ako ih korisnik ukloni. Metoda za slanje parametra prikazana je u ovome dijelu koda:

```
private async Task SendParameterAsync(string propertyName,
double value)
 {
     var parameter = new Dictionary<string, object>
{
         {propertyName, value }
     };
     await HubService.SendParamsAsync(parameter);
 }
```

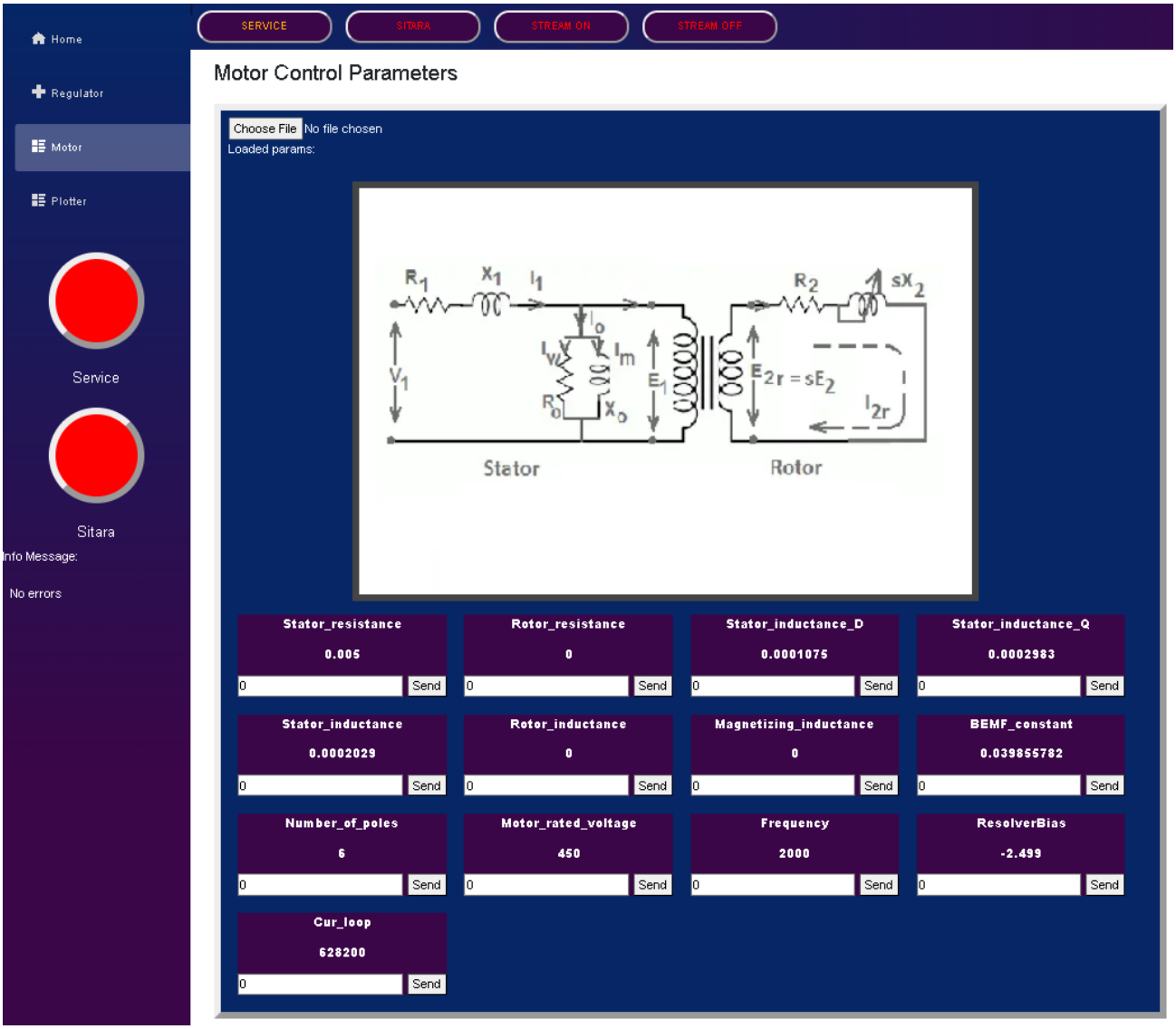
Ovdje šaljemo parametre prema servisu putem metode:

HubService.SendCommandsAsync().

Komunikacijski čvor je odgovoran za izmjenu podataka s mikrokontrolerom. Sadrži sve potrebne metode za izvršavanje slanja i zaprimanja paketa indirektno s mikrokontrolera, to jest jezgre zadužene za komunikaciju. Međusloj u komunikaciji je uz protokole implementirane na mikrokontroleru i server koji prvo procesira naredbe. Tako *HubService* instanca može postojati na više klijenata te slati zahtjeve prema serveru, kako je to opisano u uvodnome poglavlju, unutar okvira arhitekture sustava. U kontekstu spremanja parametara pozivamo univerzalnu asinkronu metodu

SendParameterAsync()

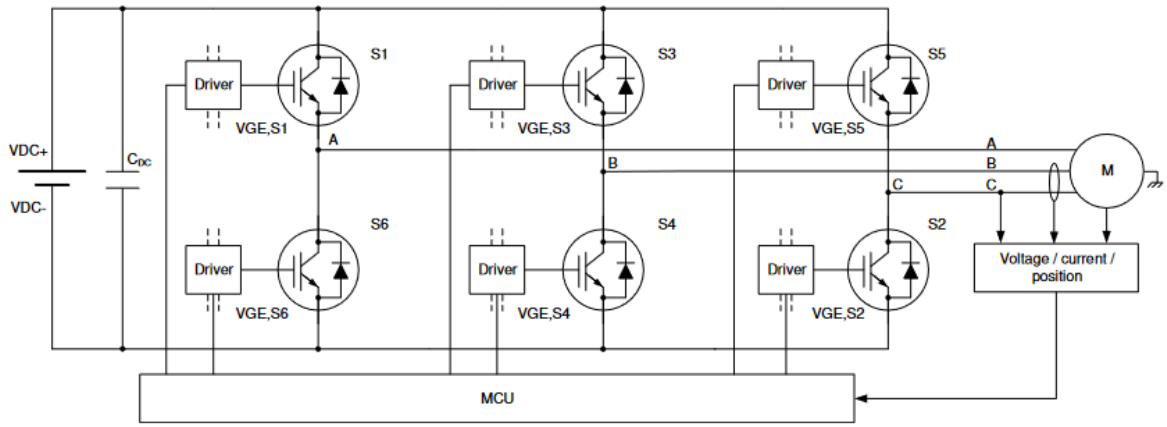
koja bilo koji tip poslane naredbe sprema u formatu rječnik koji je pogodan upravo za ovakve upotrebe. Tako možemo poslati listu parametara koje želimo proslijediti unutar jedne naredbe. Kako ne bismo opteretili mikrokontroler, listu naredbi, koje su implementirane kao lista individualnih naredbi, možemo proslijediti prema poslužitelju koji će zatim jednu po jednu slati prema mikrokontroleru maksimalno dozvoljenom brzinom koju komunikacijski protokoli mogu podnijeti bez zagušenja.



Slika 17. Konfiguracija parametara stroja

7.3. Digitalni sustav upravljanja

U sklopu regulacijske sheme prikazanoj na Slici 18. regulatori su realizirani kao programska aplikacija koja se izvršava na AM2634 procesoru. Da bi se to moglo ostvariti, to jest da bi se upravljalo na ovaj način, treba razlikovati kontinuirane i diskretne signale. Način na koji postižemo pretvorbu iz jedne vrste signala u drugu su A/D i D/A pretvornici koji metodama poput *zero* ili *first order hold* pretvaraju signal iz kontinuirane domene u diskretnu. Ove metode implementirane su u digitalnom procesoru pomoću prikladnih algoritama. Primijetimo da je *MCU – Master Control Unit* dio elektromotornoga pogona u koji su implementirani upravljački algoritmi koji moduliraju sklopke u izmjenjivačkom dijelu pretvarača na koji je spojen asinkroni stroj.



Slika 18. Shema digitalnoga sustava upravljanja 3f motorom [38]

Zadaća pobudnoga stupnja (*engl. gate driver*) je upaliti ili ugasiti sklopku u pojedinoj grani izmjenjivača. To se događa u trenutcima kada se zaprimi upravljački signal od upravljačke jedinice. Okidanja signala te korelirana stanja na sklopkama su usklađena na način kako je opisano u Tablici 3.

Tablica 3. Stanja sklopki za pozicije vektorskoga polja

Vektor	*S1*	*S2*	*S3*	*S4*	*S5*	*S6*
{000}	*OFF*	*ON*	*OFF*	*ON*	*OFF*	*ON*
{100}	*ON*	*ON*	*OFF*	*ON*	*OFF*	*OFF*
{100}	*ON*	*ON*	*ON*	*OFF*	*OFF*	*OFF*
{010}	*OFF*	*ON*	*ON*	*OFF*	*OFF*	*ON*
{011}	*OFF*	*OFF*	*ON*	*OFF*	*ON*	*ON*
{001}	*OFF*	*OFF*	*OFF*	*ON*	*ON*	*ON*
{101}	*ON*	*OFF*	*OFF*	*ON*	*ON*	*OFF*
{111}	*ON*	*OFF*	*ON*	*OFF*	*ON*	*OFF*

U vektorskoj modulaciji dostupno je osam stanja, od kojih su dva nulta vektora, dok su ostali aktivni vektori koji se koriste za primjenu potrebnoga napona na motor kako bi se generirala tražena količina momenta. Tablica prikazuje stanja u kojima su parovi prekidača S1, S6, S3 komplementarni naspram S4, S5, S2. To znači da su im stanja suprotna, to jest ne mogu biti istovremeno uključene jer bi u slučaju otvaranja obje sklopke u istoj grani bio stvoren diskretan put struje između točke višega i nižega potencijala – kratki spoj.

Prikazano je sučelje za upravljanje regulatorima struje u Q i D osi. Upravljačkom strukturom je prikazano da postoje Park i Clarke transformacijski blokovi koji su implementirani i u samome aplikacijskom algoritmu unutar procesora AM2634. Promjena parametara regulatora ili drugi blokova unutar regulacijske strukture vrši se na sličan način kao i kod parametara stroja. Unosom vrijednosti u polje te pritiskom tipke šalji poziva se metoda u programu koja dalje poziva druge metode kako bi se podatak pohranio u regulatoru te u bazi podataka za kasniju analizu. Postupak kod spremanja parametara regulatora je isti kao i kod motora. Razlika je u poslanom parametru kako bi se napravila distinkcija između parametara za motor i regulator, to jest kako bi sustav mogao prepoznati koji točno parametar povezati s prikladnim uređajem. Sam zadatak obrade podataka pada na teret poslužitelja, to jest usluge koja se izvršava na njemu, dok sa strane klijenta samo radimo zahtjeve s drukčijim parametrima prikladnima za kontrolu ili akciju koju pokušavamo napraviti ili izvesti.

U nastavku su prikazani dijelovi koda koji opisuju programske implementacije matematičkih transformacija (funkcija) koje mapiraju signale iz diskretne vremenske domene u dq koordinatni sustav radi lakše obrade i analize.

```
static inline void clarke_run(Motor_t * in)
{
    in->I_ab_A[0] = ((in->I_abc_A[0] * 2.0f) -
                     (in->I_abc_A[1] + in->I_abc_A[2])) * ONE_BY_3;
    in->I_ab_A[1] = ((in->I_abc_A[1] - in->I_abc_A[2]) * ONE_BY_SQRT3);
}

static inline void park_run(Motor_t * in)
{
    in->I_dq_A[0] = (in->I_ab_A[0] * in->Cosine) + (in->I_ab_A[1] * in->Sine);
    in->I_dq_A[1] = (in->I_ab_A[1] * in->Cosine) - (in->I_ab_A[0] * in->Sine);
}

static inline void ipark_run(Motor_t * in)
{
    in->Vout_ab_V[0] = (in->Vout_dq_V[0] * in->Cosine) -
                       (in->Vout_dq_V[1] * in->Sine_out);
    in->Vout_ab_V[1] = (in->Vout_dq_V[1] * in->Cosine) +
                       (in->Vout_dq_V[0] * in->Sine_out);
}
```

Za primjer, funkcija

*clarke_run(Motor_t * in)*

koja ne vraća nikakvu vrijednost pri kraju izvršavanja radi na način da mijenja podatkovnu strukturu izvan funkcije preko pokazivača zaprimljenoga kao ulazni parametar. Pokazivač pokazuje na strukturu koja predstavlja motor, to jest njegove parametre. Ovom transformacijom prelazimo u $\alpha\beta\gamma$ sustav te dobivamo oblik struje prema formuli za izvođenje Clarkove transformacije prikazane u nastavku u matričnoj formi:

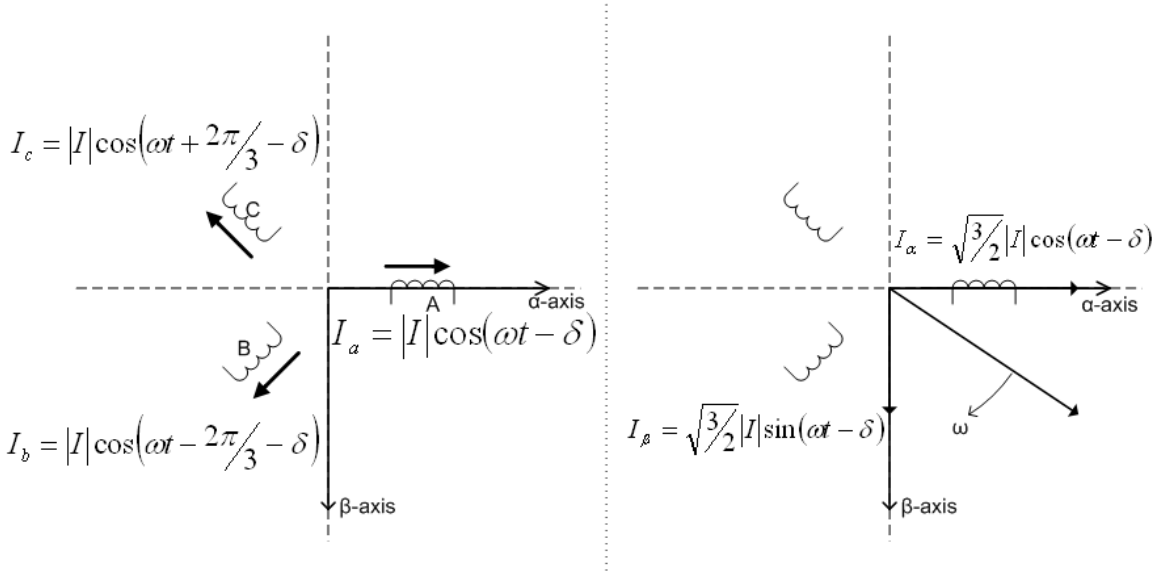
$$i_{\alpha\beta\gamma}(t) = Ti_{abc}(t) = \frac{2}{3}\begin{bmatrix} 1 & -\frac{1}{2} & -\frac{1}{2} \\ 0 & \frac{\sqrt{3}}{2} & -\frac{\sqrt{3}}{2} \\ \frac{1}{2} & \frac{1}{2} & \frac{1}{2} \end{bmatrix}\begin{bmatrix} i_a(t) \\ i_b(t) \\ i_c(t) \end{bmatrix} \quad (7.1.)$$

Gdje je γ komponenta nula zbog toga što zbroj a, b i c komponente daje nulu, pošto je promatrani sustav električno simetričan.

Inverzna transformacija se vrši na sljedeći način:

$$i_{abc}(t) = T^{-1} i_{\alpha\beta\gamma}(t) = \begin{bmatrix} 1 & 0 & 1 \\ -\frac{1}{2} & \frac{\sqrt{3}}{2} & 1 \\ -\frac{1}{2} & -\frac{\sqrt{3}}{2} & 1 \end{bmatrix} \begin{bmatrix} i_a(t) \\ i_b(t) \\ i_c(t) \end{bmatrix} \quad (7.2.)$$

Transformacija može dočarati na način da za pomaknute namotaje motora u prostoru za 120° u koordinatni sustav pozicioniramo tako da jedan od promatranih namotaja stavimo u smjeru apcise i njega nazivamo namotaj faze a, kako je prikazano na donjoj slici. Ostale namotaje crtamo pomaknute za 120° ili $\frac{2\pi}{3}$ radijana.



Slika 19. Geometrijsko predočenje transformacija [39]

Zatim se za svaki namotaj gleda njihova projekcija na α i β osi koje su u smjeru apcise i ordinate. Jasno je da će svaki namotaj imati po jednu projekciju na pojedinu os, i tako za sva tri namotaja. Usporedbom s matričnom formom transformacije vidimo da je zbog toga što smo položili namotaj

a u smjeru α osi iznos struje i_α u stvari iznos cijeloga vektora komponente a uz projekcije b i c pod istim iznosom, ali suprotnom orijentacijom. Pošto je ova pretvorba u svrhu reprezentacije tri rotirajuća vektora pomoću dva $\alpha\beta$ koordinatni sustav je mirujući te će se modul projekcije na pojedinu os mijenjati kroz vrijeme.

Domena promatranih vrijednosti nakon transformacije pogodna je za sljedeću transformaciju koja rotirajući $\alpha\beta$ sustav transformira na način da ponovno gledamo projekcije vektora $\alpha\beta$, no ovaj put na rotirajući (dq) koordinatni sustav. Time smo postigli da se amplituda promatranih varijabli i parametara ne mijenja s vremenom. Ova činjenica znatno olakšava promatranje izmjeničnih komponenti motora te u krajnosti otvara mogućnost upravljanja asinkronim motorima na veoma sličan princip kao i kod istosmjernih motora.

Parkova transformacija matematički je zapisana na sljedeći način:

$$i_{sd} = i_\alpha \cdot cos(\theta) + i_\beta \cdot sin\,(\theta) \qquad (7.3.)$$

$$i_{sq} = -i_\alpha \cdot sin(\theta) + i_\beta \cdot cos\,(\theta) \qquad (7.4.)$$

$$i_\alpha = i_{sd} \cdot cos(\theta) - i_{sq} \cdot sin\,(\theta) \qquad (7.5.)$$

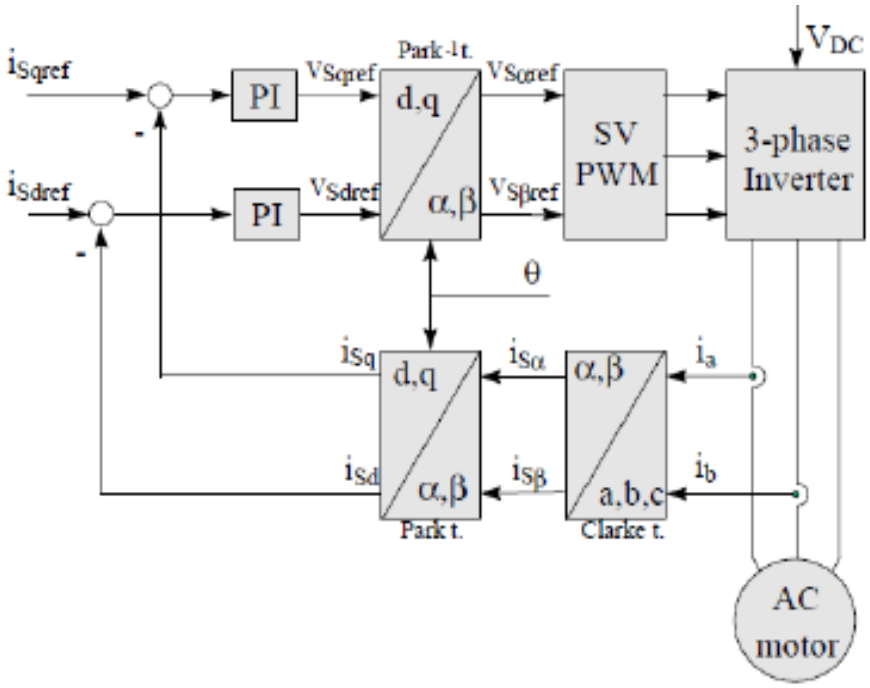
$$i_\beta = i_{sd} \cdot sin(\theta) + i_\beta \cdot cos\,(\theta) \qquad (7.6)$$

Ili u matričnoj formi:

$$\vec{i}_{dq} = \begin{bmatrix} \cos(\theta) & \sin(\theta) \\ -sin(\theta) & cos\,(\theta) \end{bmatrix} \cdot \vec{i}_{\alpha\beta} \qquad (7.7.)$$

Upravljanje orijentacijom rotorskoga toka (*engl. „Field-Oriented Control"*) ili FOC odgovara strukturi upravljanja prikazanoj na Slici 20. Ujedinjuje sve komponente i modele opisane do sada u regulacijsku strukturu te šalje pravovremene pozive pojedinim strukturama i funkcijama kako bi izvršio proračune i postavio vrijednosti pojedinoga dijela sustava. Pod vrijednostima sustava

podrazumijevamo ulazne i izlazne vrijednosti pojedinoga regulacijskog bloka prikazanoga na strukturi, u svrhu kreiranja i propagiranja upravljačkoga signala od korisnika do motora.



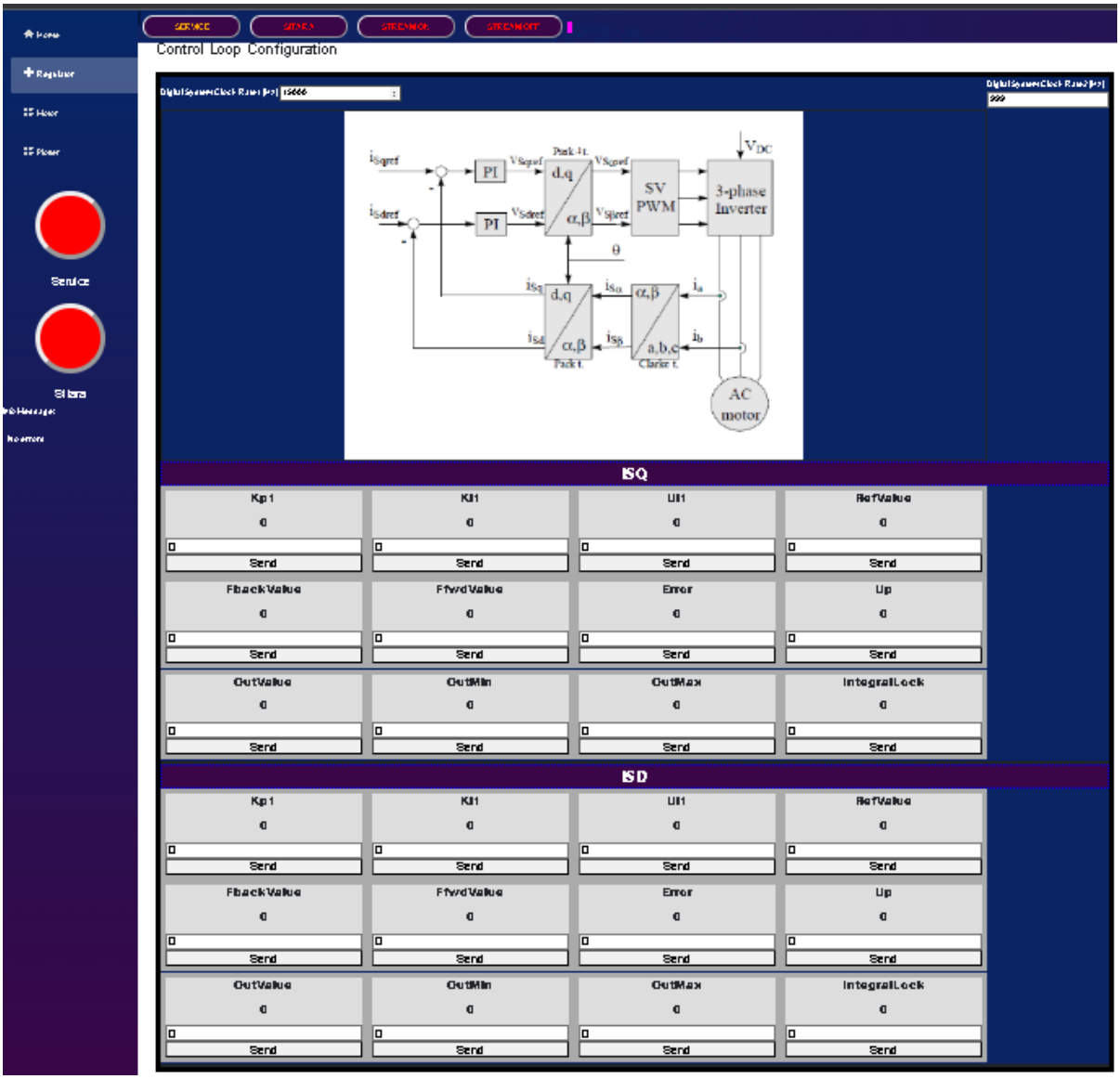
Slika 20. Shema regulacijske strukture [40]

Parametar se naziva *definicija* ulaznih vrijednosti funkcije ili metode, dok argument podrazumijeva stvarnu *vrijednost* koju šaljemo pri pozivu funkcije koju smo prethodno definirali. Definicije funkcija koje možemo pronaći u zaglavlju datoteka ili *engl. header files* sadrže sljedeći format:

```
float32_t FOC_getIdref(void);
float32_t FOC_getIqref(void);
void FOC_setIdref(float32_t value);
void FOC_setIqref(float32_t value);
```

Definirana su četiri prototipa (*engl. function prototype)* funkcija za čitanje i postavljanje referentnih vrijednosti struja u d i q osi. Kod FOC_get naredbi dohvaćamo vrijednosti te zbog toga definiramo tip funkcije kao float32_t. Parametar je u ovome slučaju tipa *void*, to jest ne postoje

ulazni parametri. FOC_set naredbe s druge strane nemaju definiran tip vrijednosti koje vraćaju, dakle prilikom poziva neće vratiti nikakvu vrijednost pozivatelju, no zato imaju definirane parametre kao float32_t. Možemo primijetiti da postoje dva smjera toka informacija, istovjetno kako blokovi sa sheme prikazani na Slici 20. imaju ulaze i izlaze iz pojedinoga bloka. Pojedini član u regulacijskoj strukturi s gornje sheme posjeduje ekvivalentni model koji predstavlja njegove funkcije u digitalnome obliku. Modeli posjeduju skup varijabli u koje pohranjujemo kvantitativna svojstva pojedinoga člana regulacijske strukture. Implementaciju na primjeru motora mogli smo vidjeti u prethodnome poglavlju. Na isti način reprezentirani su i *PI* regulatori kao blokovi regulacijske sheme. Isto tako postoje funkcije i metode koje predstavljaju opisane transformacije i proračune. Nakon što smo generirali sve potrebne podatkovne strukture, klase i njihove pripadne metode i komponente preostalo je logički povezati kako bi radile. To znači da smo povezali izlaze i ulaze pojedinih struktura koje povezuju funkcijske blokove. Kao što smo vidjeli na primjeru Clarkine transformacije, programski je to izvedeno na način da se na ulaz funkcije šalje pokazivač na strukturu kojoj želimo modificirati parametre. Parametre je moguće postaviti preko konfiguracijskoga sučelja na korisničkome dijelu aplikacije u pogledu namijenjenom za unos parametara regulatora.



Slika 21. Zaslon konfiguracije regulatora

Signal koji se primjenjuje na tri namotaja motora, kao što je poznato iz teorije o upravljanju asinkronim motorima, određuje brzinu i moment motora. Visokonaponski prekidači su najkritičnije komponente izmjenjivača jer kontroliraju tijek struje prema motoru radi generiranja pokreta, stoga se stanje prekidača kontrolira i štiti mjerenjem njihove temperature, napona i struje tijekom rada. Sklopke se upravljaju putem *MCU* jedinice i pobudnoga stupnja gornjih sklopki (HS) i donjih sklopki (LS) izmjenjivača. *PWM* signali se generiraju koristeći prostorno-vektorsku modulaciju *SVM-a*. Dok stroj radi, napon, struja i signal brzine vrtnje se mjere i povratno šalju u upravljačku strukturu kako bi se korigirao upravljački signal izmjenjivača. Jedna takva metoda

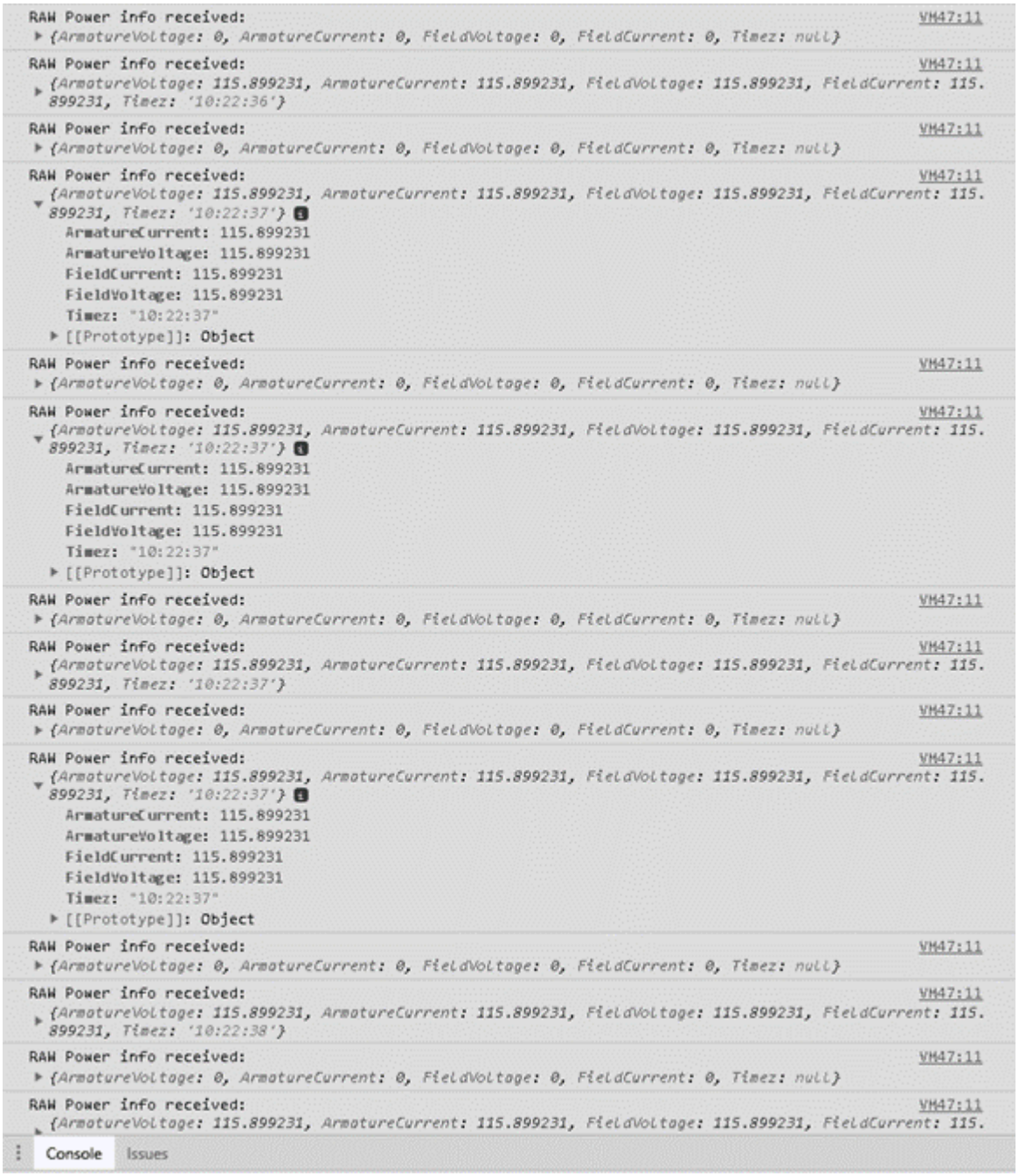
upravljanja asinkronim strojem je upravljanje korištenjem orijentacije vektorskoga toka *FOC* koja koristi dvije faze struje i poziciju za generiranje odgovarajućega referentnog vektora. Na stranici s parametrima za regulator moguće je unijeti željene vrijednosti za pojedini parametar regulatora. Pritisak na tipku uzrokuje slanje parametra prema poslužitelju koji pripadajući parametar dodjeljuje trenutno aktivnome uređaju (regulator koji je učitan u regulacijsku strukturu) te pohrani unesenu promjenu parametra u bazu podataka. Tako izvođenje omogućuje podešavanje parametara bez prekida rada motora i regulacijske strukture. To nam je omogućeno pošto je regulacijska struktura implementirana u digitalni sustav upravljanja gdje su vrijednosti pohranjene u blokove memorije kojima po želji možemo pristupati i uz pravilnu autorizaciju promijeniti vrijednost.

7.4. Prikaz grafikona i vrijednosti

U nastavku će biti opisan ostatak korisničkoga sučelja, točnije dio vezan za prikaz vrijednosti dobivenih s mikrokontrolera. Na Slici 22. je vidljiv status veze prema pojedinome sloju aplikacije u pomoćnoj traci s lijeve strane. Vidljiv je i prikaz grafikona koji prikazuju dobivene podatke iz podataka koji smo inicirali na Sitari. Svaki grafikon može se pojedinačno zaustaviti, mogu se zaustaviti svi zajedno te isto tako pokrenuti. Na apscisi pojedinoga grafikona vidljivo je trenutno vrijeme. Svi grafikoni koriste istu vremensku skalu, pa je vrijednosti na njima moguće paralelno uspoređivati.

Slika 22. Stranica za prikaz grafikona

Na grafikonima su dostupni prikazi brzine vrtnje, napona izvora te struje u d i q osi. Model koji koristimo pri slanju podataka ispunjen je svim podacima koje primamo s kontrolera motora. Na ovoj stranici mogu se implementirati dodatni grafikoni sa željenim vrijednostima sustava. Odabrani su prikazani zbog jednostavnosti te je uzeto u obzir da su ovo neki primarni parametri koji za demonstracijsku svrhu mogu poslužiti toj namjeni. Referenca motora konfigurirana je da se kreće po uzlazno-silaznoj rampi također za demonstracijske svrhe. Ona je prikazana u postotnim vrijednostima, to jest kao koeficijent s rasponom vrijednosti od nule do jedan. Grafikon sa strujama pokazuje vrijednosti struje u d i q osi. Pošto nemamo simuliranoga tereta na motoru, on se vrti u praznome hodu. Stoga prilikom ubrzanja i usporavanja po sporoj rampi nema primjetne razlike između referentne i mjerene veličine te struja u d osi oscilira oko nule za ugađanje brzine vrtnje motora.



Slika 23. Podatci zaprimljeni pomoću IJSR-a

Da bismo omogućili ažuriranja grafikona u stvarnome vremenu, pretplatili smo se na događaje (*events*) koje *SignalR* čvor šalje kada se podatci promijene. Ove promjene smo zatim ažurirali u *chartData* objektu koji je vezan uz naš grafikon.

Primjer koda za pretplatu na promjene podataka:

```
    private async void OnSitaraConnectionStatusChanged(object sender, int
isSitaraConnectionOpen)
    {
        await InvokeAsync(() =>
           {
               statusInfoList[1].IsConnected = isSitaraConnectionOpen;
               statusInfoList[1].Status = isSitaraConnectionOpen > 0 ?
"ConnectedSitara" : "DisconnectedSitara";
               SaveConnectionStatus(statusInfoList[1].IsConnected);
               StateHasChanged(); // Update the UI
           });
    }
```

Kako bismo koristili *Chart* komponentu unutar aplikacije, dodali smo potrebne reference na *ChartJs* biblioteku. Također smo konfigurirali *chartData* i *chartOptions* klase kako bismo prilagodili izgled i ponašanje grafikona.

Primjer koda za konfiguraciju grafikona:

```
private LineConfig chartData;
private LineOptions chartOptions;
protected override async Task OnInitializedAsync()
{
    chartData = new LineConfig
    {
        Labels = new List<string>(),
        Datasets = new List<LineDataset> { new LineDataset<int> { Label = "Speed", Fill
= false, BorderColor = "rgba(75, 192, 192, 1)" } }
    };
    chartOptions = new LineOptions
    {
        Responsive = true,
        MaintainAspectRatio = false,
        Scales = new Scales
        {
            X = new CategoryScale { Display = true, Title = new ScaleTitle { Display =
true, Text = "Time" } },
            Y = new LinearScale { Display = true, Title = new ScaleTitle { Display =
true, Text = "Speed (RPM)" } }
        }
    };
    await SubscribeToMotorDataChanges();}
```

Ovim koracima smo omogućili praćenje podataka u stvarnome vremenu i prikazivanje istih na grafikonima u aplikaciji. Korištenje *Chart.js* u kombinaciji sa *SignalR* protokolom omogućilo je stvaranje dinamičkoga i interaktivnoga sučelja koje se automatski ažurira kako bi odražavalo promjene sustava u stvarnome vremenu.

Chart.js je *JavaScript* biblioteka za stvaranje dinamičkih grafikona i dijagrama u mrežnim aplikacijama. U kontekstu zadatka *Chart.js* se koristi za vizualizaciju podataka i prikazivanje grafičkih prikaza različitih statistika i metrika. U nastavku je dio koda zadužen za reagiranje na događaj zaprimanja podataka o trenutnim vrijednostima varijabli otipkanih iz regulacijske strukture. Kao i do sada, obrađujemo *JSON* format te iz njega očitavamo parametre i dodjeljujemo nove vrijednosti u promatrani skup podataka.

```
window.onPowerInfoReceived = (powerInfoJson) => {
        const powerInfo = JSON.parse(powerInfoJson);
        myChart1.data.datasets.forEach((dataset, datasetIndex) => {
            // Update dataset data based on powerInfoJson
            let newYValue;
            let newXValue;
                if (dataset.label === 'Armature') {
                    newYValue = powerInfo.ArmatureVoltage ;
                    newXValue = Date.now();
                }
                else if (dataset.label === 'Field') {
                    newYValue = powerInfo.FieldVoltage;
                    newXValue = Date.now();
                }                dataset.data.push({
                    x: newXValue,
                    y: newYValue
                });
        });
```

Kako bismo koristili *Chart.js* skriptu u projektu, potrebno je uključiti biblioteku u projekt. To možemo učiniti na različite načine, ali je upotrijebljeno uključivanje *Chart.js* datoteka putem *CDN* (*Content delivery network*) mreže u *XML* stranici projekta pripadnim naredbama.
Nakon što je *Chart.js* biblioteka uključena u projekt, kreiran je prikaz i komponenta u aplikaciji koja će koristiti ovu biblioteku.

Na primjer, možemo kreirati komponentu koja će biti odgovorna samo za stvaranje i prikazivanje grafikona koristeći *Chart.js*. naredbu i skriptu. Kreirana je nova stranica unutar aplikacije samo za grafički prikaz otipkanih vrijednosti u periodu 100 ms, za vrijednosti napona struje armature, snage, momenta te brzine vrtnje.

U toj komponenti možemo definirati parametre grafikona, poput vrste grafikona (npr. linija, stupci, kružnica itd.), stilova, podataka koje želimo prikazati i ostale opcije. *Chart.js* će zatim generirati grafički prikaz na temelju tih parametara. U kodu u nastavku prikazano je kako podesiti pojedini

grafikon. Pošto se radi s *JavaScript* jezikom te tipovima podataka čiji su format i zapis standardizirani, korištenje *JSON* strukture podataka tijekom projekta gdje je god to moguće uvelike olakšava tijek i transformaciju podataka na pojedinoj razini aplikacije te naposljetku i sam ispis i prikaz vrijednosti.

Unutar varijable *data* definiramo dva skupa podataka, jedan za referentne, a drugi za mjerene veličine. Oni su ovdje samo inicijalizirani i najavljeni za korištenje, dok se stvarni niz podataka i vrijednosti parametara dohvaća s mikrokontrolera te se zatim obrađuje, pretvara u *JSON* format i dostavlja i korištenjem *IJSR* rutine.

Kod u nastavku opisuje implementaciju navedene strukture:

```
<script>
    data = {
        datasets: [{
            label: 'Reference',
            data: [],
            borderWidth: 2
        }
            , {
            label: 'Measured',
            data: [],
            borderWidth: 2
        }]
    };

    config = {
        type: 'line',
        data,
        options: {
            plugins: {
                streaming: {
                    ttl: 10000,
                    refresh: 200,
                }
            },
            scales: {
                x: {
                    type: 'realtime',
                    realtime: {
                        onRefresh: chart => {
                         …
                        }
                    }
                },
                y: {
                    beginAtZero: true
                }
            }
        }
    };
    ctx1 = document.getElementById('voltages');
    myChart1 = new Chart(ctx1,
        config
    );
```

Dio aplikacije predstavlja i upotreba *CSS*-a. To je dekorativni jezik koji služi za vizualne modifikacije *HTML* elemenata na stranici. Korišten je, na primjer, za promjenu boje dugmadi prilikom stanja prekida veze između slojeva aplikacije ili za promjenu boje pozadine.

Ovdje se podatci pretvaraju iz binarnoga formata u oblik pogodan za daljnju analizu i prikazivanje grafikona. U nastavku je dan uvid kako *TCPClient* klasa i *GetStream* metoda djeluju. *TCPClient* se koristi za uspostavu i upravljanje *TCP* veze s mikrokontrolerom. *GetStream* je metoda koja otvara tijek podataka na toj vezi. Dakle, *GetStream* metoda kreira komunikacijski kanal. Podatci se preko ovoga toka šalju i primaju. Za privremenu pohranu podataka prilikom prijenosa definiran je međuspremnik za prijem podataka veličine 1024 bajta:

```
byte[] receivedBytes = new byte[1024];
```

Veličina međuspremnika nije nasumična. Odabrana je prema potrebama sustava. Međuspremnik djeluje kao spremnik za prijem podataka. Što je on veći, to više podataka može biti primljeno u jednom paketu, ali to znači i veću potrošnju memorije. Nasuprot tome, premali međuspremnik može rezultirati isprekidanim prijemom i nepotpunim podacima. Veličina od 1024 bajta čini kompromis između efikasnosti i brzine prijenosa podataka.

U slučaju da mikrokontroler šalje informacije o grafikonu u obliku niza brojeva s razmacima, na primjer (25, 30, 35, 40, 45), ovo su vrijednosti grafikona koje će sustav prihvatiti i obraditi. Kada se koriste *TCPClient* i *GetStream*, međuspremnik *receivedBytes* se koristi za pohranu ovih podataka iz toka podataka. Tok podataka će sadržavati binarni format ovih brojeva. U našem slučaju, to će biti niz bajtova.

```
int bytesRead = await _tcpClient.GetStream().ReadAsync(receivedBytes, 0,
                                                receivedBytes.Length);
```

Ova linija koda čita podatke iz toka i pohranjuje ih u *receivedBytes* međuspremnik. Nakon čitanja, imamo informaciju o broju bajtova koji su pročitani, što je važno jer se ne zna unaprijed koliko će podataka biti poslano.

Kako bismo pretvorili binarni format u čitljiv format, koristimo *ASCII* kodiranje. *ASCII* (*American Standard Code for Information Interchange*) je standardna kodna shema koja mapira brojeve (0 - 127) na odgovarajuće znakove kao što su slova, brojevi i simboli. Na primjer, *ASCII* vrijednost za slovo "A" je 65. Kako bi se primljeni binarni podatci pretvorili u *ASCII*, koristimo sljedeću liniju koda:

string receivedData = Encoding.ASCII.GetString(receivedBytes, 0, bytesRead);

Ovdje se koristi *Encoding.ASCII* klasa da bi se binarni podaci pretvorili u tekstualni oblik. *GetString()* metoda spomenute klase interpretira niz bajtova koristeći *ASCII* mapiranje. Nakon ovoga koraka, *receivedData* će sadržavati čitljive tekstualne podatke koje je moguće dalje analizirati i koristiti za stvaranje grafikona.

Prilikom enkodiranja i dekodiranje moguće su pogreške u prijenosu. Ako mikrokontroler šalje podatke u formatu koji nije u skladu s *ASCII* standardom, doći će do pogrešaka dekodiranja. Da bismo obradili takve greške, potrebno je uključiti odgovarajući mehanizam obrade pogrešaka u kod kako bismo izbjegli neočekivano prekidanje programa ili pogrešne interpretacije podataka. U tu svrhu provedeno je i dodatno naknadno ispitivanje svake komande kako bi se potvrdilo ispravno djelovanje napisanih metoda i funkcija te samo ponašanje sustava.

Nakon što su primljeni podatci u obliku teksta putem *TCP* veze i *ASCII* dekodirani, slijedi proces obrade tih podataka kako bi se stvorila instanca klase *PowerInfo* koja će sadržati svojevrsni snimak trenutnoga stanja na uređaju. Ta klasa, kako je navedeno u opisu korištenih struktura podataka, sadrži informacije o trenutnim stanjima napona i struje na uzbudi i armaturi motora. Ovaj korak je ključan jer omogućuje sustavu razumijevanje i organiziranje podataka za daljnju analizu i prikazivanje grafikona.

Primljeni tekstualni podaci koji sadrže informacije o snazi grafikona su pohranjeni u varijablu *receivedData*. Za transformaciju i čitanje podataka iz tekstualnih vrijednosti koristi se *JsonSerializer* koji je dio *System.Text.Json* biblioteke. Ova biblioteka omogućuje obradu *JSON* (*JavaScript Object Notation*) podataka, što je popularan način za organizaciju strukturiranih podataka, također objašnjeno u prethodnim poglavljima.

U sljedećoj liniji koda koristimo *JsonSerializer* za pretvorbu tekstualnih podataka u *PowerInfo* objekt:

PowerInfo powerData = ParsePowerInfo(receivedData);

Metoda *ParsePowerInfo* se koristi za dekodiranje i izvlačenje podataka iz primljenih tekstualnih podataka u *PowerInfo* objekt. Ta metoda izgleda ovako:

```
private PowerInfo ParsePowerInfo(string receivedData)
{
    PowerInfo powerInfo = new PowerInfo();
    try
    {
        // Koristimo System.Text.Json.JsonSerializer za deserializaciju JSON
teksta
        powerInfo = JsonSerializer.Deserialize<PowerInfo>(receivedData);
    }
    catch (Exception ex)
    {
        // Obrada greške ako parsiranje ne uspije
        Console.WriteLine($"Greška prilikom parsiranja primljenih podataka:
{ex.Message}");
    }
    return powerInfo;
}
```

Kao što je vidljivo, ovdje se koristi *JsonSerializer.Deserialize* metoda kako bi prepoznala i pročitala *JSON* podatke iz teksta i pretvorila ih u *PowerInfo* objekt koji koristimo kao objekt koji šaljemo dalje, a sadrži informacije o pogonu sustava. Ako primljeni tekst nije ispravno oblikovan kao *JSON*, doći će do pogrešaka pri parsiranju. Stoga je bitno uključiti mehanizam za obradu tih pogrešaka kako bi se nosili s neočekivanim situacijama. U prethodno navedenom primjeru, ako parsiranje ne uspije, pogreška će biti uhvaćena i prikazana u konzoli, što omogućava dijagnostiku i rješavanje problema.

8. ZAKLJUČAK

Rad je predstavio istraživanje iz područja ugradbenih sustava, mrežnih tehnologija i razvoja korisničkoga sučelja, rezultirajući algoritmom za upravljanje asinkronim strojem i korisničkim sučeljem za izmjenu podataka s mikrokontrolerom. Objašnjava korištenje mikrokontrolera Texas Instruments AM2634, uz pomoć TCP i Ethernet komunikacijskih protokola te protokola za interprocesorsku komunikaciju. Napravljen je nacrt izgleda korisničkoga sučelja prema dogovorenim zahtjevima i funkcionalnostima.

Kreirana je početna stranica sučelja po uputama iz nacrta. Dodane su grafičke komponente koje predstavljaju tipke i polja za pružanje kontrole korisniku. Kontrolama dodane su pripadne metode i funkcije koje omogućavaju prijenos informacija na ostatak sustava. Razvijen je poslužiteljski dio sustava. Pritom su definirane funkcije i protokoli za mrežnu komunikaciju ostalih dijelova sustava. Korišteni su mrežni protokoli te su razvijene metode i funkcije čija je svrha razmjena podataka. Metode su razvijene pomoću tehnika paralelnoga izvođenja programa te asinkronih metoda, kako bi sustav u što kraćem vremenu reagirao na promjene i zahtjeve. Zatim je postavljena mrežna komunikacija između poslužitelja i mikrokontrolera. Testirane su brzine prijenosa veze, tip i količina podataka koje je moguće slati putem ove veze. Razvoj je paralelno vršen na strani poslužitelja te na strani mikrokontrolera jer podrazumijeva komunikaciju između dva podsustava pisanih u različitim programskim jezicima te izvođenih na različitim procesorskim jedinicama. Prilikom uspješnoga prijenosa podataka od korisničkoga sučelja do mikrokontrolera, razvijena je funkcionalnost interprocesorske komunikacije.

U konačnici su razvijeni i funkcijski blokovi za regulaciju parametara d i q osi asinkronoga stroja. Razvijen je model stroja kako bi se mogle simulirati vrijednosti prilikom rada sustava te kako bismo mogli dobiti vizualnu reprezentaciju podataka na strani korisničkoga sučelja. Razvijeno programsko rješenje pruža platformu za daljnji razvoj i implementacije dodatnih funkcionalnosti. Moguće je korištenje i s ostalim tipovima električnih strojeva i aktuatora kao što su koračni ili istosmjerni strojevi. Planirani su dodatci i proširenja za prikaz grafikona, poput uvećavanja prikaza, bržega vremena osvježenja grafikona, odabira komponenti za prikaz na grafikonu (odabir iz izbornika s dostupnim komponentama za prikaz, na primjer, prikaži struju i napon armature, ispis na zajedničkom, objedinjenom grafikonu).

9. LITERATURA

[1] »Transmission Control Protocol,« [Mrežno]. https://en.wikipedia.org/wiki/Transmission_Control_Protocol. [Zadnji pristup 8 11 2023].

[2] »Internet Protocol,« [Mrežno]. https://en.wikipedia.org/wiki/Internet_Protocol. [Zadnji pristup 1 11 2023].

[3] »Ethernet,« [Mrežno]. https://en.wikipedia.org/wiki/Ethernet. [Zadnji pristup 1 11 2023].

[4] »Docker,« [Mrežno]. https://www.docker.com/resources/what-container/. [Zadnji pristup 24 12 2023].

[5] »Git,« [Mrežno]. https://en.wikipedia.org/wiki/Git. [Zadnji pristup 25 12 2023].

[6] »Visual Studio,« [Mrežno]. https://learn.microsoft.com/en-us/visualstudio/get-started/visual-studio-ide?view=vs-2022. [Zadnji pristup 17 9 2023].

[7] »Code Composer Studio,« [Mrežno]. https://www.ti.com/tool/CCSTUDIO. [Zadnji pristup 19 7 2023].

[8] »SQL,« [Mrežno]. https://learn.microsoft.com/en-us/sql/sql-server/what-is-sql-server?view=sql-server-ver16. [Zadnji pristup 17 11 2023].

[9] »Dockerfile,« [Mrežno]. https://docs.docker.com/engine/reference/builder/. [Zadnji pristup 20 12 2023].

[10] ».NET,« [Mrežno]. https://en.wikipedia.org/wiki/.NET_Framework. [Zadnji pristup 20 11 2023].

[11] »Bitbucket,« [Mrežno]. https://support.atlassian.com/bitbucket-cloud/docs/build-test-and-deploy-with-pipelines/. [Zadnji pristup 15 9 2023].

[12] »Postman,« [Mrežno]. https://www.postman.com/product/what-is-postman/. [Zadnji pristup 25 12 2023].

[13] »Swagger,« [Mrežno]. https://swagger.io/solutions/api-documentation/. [Zadnji pristup 11 11 2023].

[14] »SignalR,« [Mrežno]. https://learn.microsoft.com/en-us/aspnet/core/signalr/introduction?view=aspnetcore-8.0&WT.mc_id=dotnet-35129-website. [Zadnji pristup 4 10 2023].

[15] »WebSocket,« [Mrežno]. : https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API. [Zadnji pristup 17 11 2023].

[16] »JSON,« [Mrežno]. : https://www.json.org/json-en.html. [Zadnji pristup 11 9 2023].

[17] [Mrežno]. https://www.ti.com/tool/SYSCONFIG. [Zadnji pristup 25 12 2023].

[18] »MAC,« [Mrežno]. https://en.wikipedia.org/wiki/Medium_access_control. [Zadnji pristup 25 12 2023].

[19] »DHCP,« [Mrežno]. https://learn.microsoft.com/en-us/windows-server/networking/technologies/dhcp/dhcp-top. [Zadnji pristup 25 12 2023].

[20] »Redis,« [Mrežno]. https://redis.com/ebook/part-1-getting-started/chapter-1-getting-to-know-redis/1-1-what-is-redis/1-1-1-redis-compared-to-other-databases-and-software/. [Zadnji pristup 1 9 2023].

[21] »pipeline,« [Mrežno]. https://en.wikipedia.org/wiki/Pipeline_(software). [Zadnji pristup 23 12 2023].

[22] »XAML,« [Mrežno]. https://en.wikipedia.org/wiki/Extensible_Application_Markup_Language. [Zadnji pristup 24 12 2023].

[23] »Netvscore,« [Mrežno]. https://learn.microsoft.com/en-us/aspnet/core/fundamentals/choose-aspnet-framework?view=aspnetcore-8.0. [Zadnji pristup 25 12 2023].

[24] »Reflexion,« [Mrežno]. https://learn.microsoft.com/en-us/dotnet/csharp/advanced-topics/reflection-and-attributes/. [Zadnji pristup 14 10 2023].

[25] »IJSRuntime,« [Mrežno]. https://learn.microsoft.com/en-us/dotnet/api/microsoft.jsinterop.ijsruntime?view=aspnetcore-8.0. [Zadnji pristup 25 12 2023].

[26] »Events,« [Mrežno]. https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/events/. [Zadnji pristup 20 10 2023].

[27] »Taskthread,« [Mrežno]. https://learn.microsoft.com/en-us/dotnet/api/system.threading.tasks.task?view=net-8.0. [Zadnji pristup 18 8 2023].

[28] »taskthreadimg,« [Mrežno]. https://realpython.com/python-concurrency/. [Zadnji pristup 18 8 2023].

[29] »issuewithtcp,« [Mrežno]. https://learn.microsoft.com/en-US/previous-versions/troubleshoot/windows/win32/data-segment-tcp-winsock. [Zadnji pristup 16 9 2023].

[30] »synack,« [Mrežno]. https://github.blog/wp-content/uploads/2016/07/tcp-3whs.png. [Zadnji pristup 25 12 2023].

[31] »gpio,« [Mrežno]. https://www.ti.com/lit/ug/spruj17e/spruj17e.pdf?ts=1703478201458. [Zadnji pristup 25 12 2023].

[32] »dma,« [Mrežno]. https://www.ti.com/lit/ug/slau395f/slau395f.pdf?ts=1703528028343&ref_url=https%253A%252F%252Fwww.google.com%252F. [Zadnji pristup 25 12 2023].

[33] »ipcimg,« [Mrežno]. https://software-dl.ti.com/jacinto7/esd/processor-sdk-rtos-jacinto7/07_00_00_11/exports/docs/psdk_rtos_auto/docs/user_guide/developer_notes_ipc.html. [Zadnji pristup 16 9 2023].

[34] »float32,« [Mrežno]. https://en.wikipedia.org/wiki/IEEE_754. [Zadnji pristup 1 9 2023].

[35] »windup,« [Mrežno]. https://www.fer.unizg.hr/_download/repository/RUS_predavanje_19.pdf. [Zadnji pristup 24 12 2023].

[36] »Method parameters,« [Mrežno]. https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/method-parameters. [Zadnji pristup 4 1 2024].

[37] ».NET Dictionary,« [Mrežno]. https://learn.microsoft.com/it-it/dotnet/api/system.collections.generic.dictionary-2.keys?view=net-8.0. [Zadnji pristup 1 4 2024].

[38] »dsp,« [Mrežno]. https://www.ti.com/lit/ml/slyp888/slyp888.pdf?ts=1703509741438. [Zadnji pristup 14 12 2023].

[39] »dqalfabeta,« [Mrežno]. https://upload.wikimedia.org/wikipedia/commons/9/9f/AlphaBeta_geometric_interpretation.gif. [Zadnji pristup 18 12 2023].

[40] »FOC,« [Mrežno]. https://www.researchgate.net/figure/The-field-oriented-control-scheme-without-speed-controller-part-Texas-Instruments_fig1_242070917. [Zadnji pristup 25 12 2023].

SAŽETAK

U radu je istražena implementacija digitalnih sustava upravljanja. Naglasak je bio na razvoju programskih rješenja i algoritama prilagođenih izvođenju na mikrokontroleru naspram klasičnih PLC jedinica. Korišten je *Texas Instruments AM2634* četverojezgreni mikrokontroler. Na mikrokontroleru su primijenjene tehnike paralelnoga izvođenja algoritama na pojedinoj jezgri mikrokontrolera sa svrhom odvajanja odgovornosti izvođenja. Jedna jezgra služi za komunikaciju, dok druga vodi računa o upravljanju električnim strojem. Izrađen je i konfiguriran programski paket za izvođenje na računalnom mrežnom poslužitelju u svrhu distribucije podataka s mikrokontrolera prema korisnicima pomoću *C#* jezika i pripadajućega *.NET* programskoga okvira. Komponente sustava pokretane su u izoliranome okruženju za izvođenje pomoću *Docker* platforme za virtualizaciju sustava. Za kreiranje korisničkoga sučelja korišten je *Blazor* mrežni okvir tvrtke *Microsoft*. Tijekom izvođenja algoritama prikupljeni podatci spremaju se u *SQL* bazu podataka. Na korisničkome sučelju, dostupnom putem svih popularnijih internetskih preglednika, izrađene su upravljačke kontrole za slanje osnovnih naredbi za upravljanje motorom. Na korisničkome sučelju moguće je prikazivati pristigle podatke u tekstualnome i slikovnome obliku. Podatci su s mikrokontrolera poslani mrežnim putem prema korisničkome sučelju gdje se pokušalo ostvariti što kraće vrijeme slanja težeći vremenima izvođenja sustava u stvarnome vremenu. Nakon zaprimanja podataka preko mrežnoga poslužitelja komunikacijskih usluga, na korisničkome sučelju podatci su uz pomoć *ChartJS javascript* biblioteka prikazani u grafikonima. Sustav je napravljen da pruža bazu za kasnije jednostavno dodavanje podrške novim proširenjima za sve tipove električnih strojeva te je moguća implementacija popularnih biblioteka koje podržavaju strojno učenje prilikom upravljanja strojem.

Ključne riječi: Digitalni sustavi upravljanja, mikrokontroler AM2634, Texas Instruments, Code Composer Studio, Microsoft, C, C#, .NET, javascript, Docker, interprocesorska komunikacija, paralelno izvođenje, dretve, mrežni protokoli, TCP, IP, Ethernet, konfiguracija mrežnih postavki, baze podataka, upravljanjem orijentacijom polja električnog stroja.

ABSTRACT

The paper explored the implementation of digital control systems, with an emphasis on developing software solutions and algorithms tailored for execution on a microcontroller compared to traditional *PLC* units. The Texas Instruments AM2634 quad-core microcontroller was used in this study. Parallel algorithm execution techniques were applied on individual microcontroller cores to separate the responsibilities of execution. One core was dedicated to communication, while another core handled the control of the electrical machine. A software package was developed and configured to run on a computer network server for distributing data from the microcontroller to users using the C# language and the corresponding .NET framework. The system components were run in an isolated execution environment using the Docker platform for system virtualization. The Microsoft Blazor web framework was used to create the user interface. During algorithm execution, collected data was stored in an SQL database. The user interface, accessible via popular web browsers, featured control elements for sending basic commands to control the motor. The user interface could display received data in textual and graphical formats. Data from the microcontroller were transmitted over the network to the user interface, where efforts were made to achieve minimal transmission times, aiming for real-time system performance. Upon receiving data through the communication service network server, the user interface used ChartJS JavaScript libraries to present the data in graphs. The system was designed to provide a foundation for easily adding support for new extensions for all types of electrical machines. Implementation of popular machine learning libraries for machine control is also possible.

Keywords: Digital control systems, AM2634 microcontroller, Texas Instruments, Code Composer Studio, Microsoft, C, C#, .NET, JavaScript, Docker, interprocess communication, parallel execution of tasks, threads, network protocols, TCP, IP, Ethernet, network configuration, databases, field oriented control.

DODATAK

9.1. Radna memorija

Tablica 3. Mapa memorije Sitara mikrokontrolera

CSL_HSM_ROM_U_BASE	(0x20000000ul)
CSL_HSM_SEC_ROM_U_BASE	(0x20010000ul)
CSL_HSM_RAM_U_BASE	(0x20020000ul)
CSL_MCRC0_U_BASE	(0x35000000ul)
CSL_STM_STIM_U_BASE	(0x39000000ul)
CSL_HSM_SOC_CTRL_U_BASE	(0x40000000ul)
CSL_MPU_L2OCRAM_BANK0_U_BASE	(0x40020000ul)
CSL_MPU_L2OCRAM_BANK1_U_BASE	(0x40040000ul)
CSL_MPU_L2OCRAM_BANK2_U_BASE	(0x40060000ul)
CSL_MPU_L2OCRAM_BANK3_U_BASE	(0x40080000ul)
CSL_MPU_R5SS0_CORE0_AXIS_U_BASE	(0x400A0000ul)
CSL_MPU_R5SS0_CORE1_AXIS_U_BASE	(0x400C0000ul)
CSL_MPU_R5SS1_CORE0_AXIS_U_BASE	(0x400E0000ul)
CSL_MPU_R5SS1_CORE1_AXIS_U_BASE	(0x40100000ul)
CSL_MPU_HSM_DTHE_U_BASE	(0x40120000ul)
CSL_MPU_MBOX_SRAM_U_BASE	(0x40140000ul)
CSL_MPU_QSPI0_U_BASE	(0x40160000ul)
CSL_MPU_SCRM2SCRP0_U_BASE	(0x40180000ul)
CSL_MPU_SCRM2SCRP1_U_BASE	(0x401A0000ul)
CSL_MPU_R5SS0_CORE0_AHB_U_BASE	(0x401C0000ul)
CSL_MPU_R5SS0_CORE1_AHB_U_BASE	(0x401E0000ul)
CSL_MPU_R5SS1_CORE0_AHB_U_BASE	(0x40200000ul)
CSL_MPU_R5SS1_CORE1_AHB_U_BASE	(0x40220000ul)
CSL_MPU_HSM_U_BASE	(0x40240000ul)
CSL_HSM_SOC_PCR_U_BASE	(0x40F78000ul)
CSL_HSM_STC_U_BASE	(0x40F78C00ul)
CSL_HSM_PBIST_U_BASE	(0x40F79000ul)

CSL_HSM_ECC_AGGR_U_BASE	(0x40F79400ul)
CSL_HSM_MBOX_SRAM_U_BASE	(0x44000000ul)
CSL_HSM_SEC_MGR_U_BASE	(0x46000000ul)
CSL_HSM_SEC_RAM_U_BASE	(0x46050000ul)
CSL_HSM_CTRL_U_BASE	(0x47000000ul)
CSL_HSM_TPCC0_U_BASE	(0x47020000ul)
CSL_HSM_TPTC00_U_BASE	(0x47040000ul)
CSL_HSM_TPTC01_U_BASE	(0x47060000ul)
CSL_HSM_PCR_U_BASE	(0x47F78000ul)
CSL_HSM_RTI0_U_BASE	(0x47F78C00ul)
CSL_HSM_WDT0_U_BASE	(0x47F78D00ul)
CSL_HSM_DCC0_U_BASE	(0x47F79000ul)
CSL_HSM_ESM_U_BASE	(0x47F79400ul)
CSL_HSM_DMT0_U_BASE	(0x47F79800ul)
CSL_HSM_DMT1_U_BASE	(0x47F79900ul)
CSL_ICSSM0_INTERNAL_U_BASE	(0x48000000ul)
CSL_ICSSM0_ECC_U_BASE	(0x48100000ul)
CSL_QSPI0_U_BASE	(0x48200000ul)
CSL_MMC0_U_BASE	(0x48300000ul)
CSL_GPMC0_CFG_U_BASE	(0x48400000ul)
CSL_CONTROLSS_G0_EPWM0_U_BASE	(0x50000000ul)
CSL_CONTROLSS_G0_EPWM1_U_BASE	(0x50001000ul)
CSL_CONTROLSS_G0_EPWM2_U_BASE	(0x50002000ul)
CSL_CONTROLSS_G0_EPWM3_U_BASE	(0x50003000ul)
CSL_CONTROLSS_G0_EPWM4_U_BASE	(0x50004000ul)
CSL_CONTROLSS_G0_EPWM5_U_BASE	(0x50005000ul)
CSL_CONTROLSS_G0_EPWM6_U_BASE	(0x50006000ul)
CSL_CONTROLSS_G0_EPWM7_U_BASE	(0x50007000ul)
CSL_CONTROLSS_G0_EPWM8_U_BASE	(0x50008000ul)
CSL_CONTROLSS_G0_EPWM9_U_BASE	(0x50009000ul)

CSL_CONTROLSS_G0_EPWM10_U_BASE	(0x5000A000ul)
CSL_CONTROLSS_G0_EPWM11_U_BASE	(0x5000B000ul)
CSL_CONTROLSS_G0_EPWM12_U_BASE	(0x5000C000ul)
CSL_CONTROLSS_G0_EPWM13_U_BASE	(0x5000D000ul)
CSL_CONTROLSS_G0_EPWM14_U_BASE	(0x5000E000ul)
CSL_CONTROLSS_G0_EPWM15_U_BASE	(0x5000F000ul)
CSL_CONTROLSS_G0_EPWM16_U_BASE	(0x50010000ul)
CSL_CONTROLSS_G0_EPWM17_U_BASE	(0x50011000ul)
CSL_CONTROLSS_G0_EPWM18_U_BASE	(0x50012000ul)
CSL_CONTROLSS_G0_EPWM19_U_BASE	(0x50013000ul)
CSL_CONTROLSS_G0_EPWM20_U_BASE	(0x50014000ul)
CSL_CONTROLSS_G0_EPWM21_U_BASE	(0x50015000ul)
CSL_CONTROLSS_G0_EPWM22_U_BASE	(0x50016000ul)
CSL_CONTROLSS_G0_EPWM23_U_BASE	(0x50017000ul)
CSL_CONTROLSS_G0_EPWM24_U_BASE	(0x50018000ul)
CSL_CONTROLSS_G0_EPWM25_U_BASE	(0x50019000ul)
CSL_CONTROLSS_G0_EPWM26_U_BASE	(0x5001A000ul)
CSL_CONTROLSS_G0_EPWM27_U_BASE	(0x5001B000ul)
CSL_CONTROLSS_G0_EPWM28_U_BASE	(0x5001C000ul)
CSL_CONTROLSS_G0_EPWM29_U_BASE	(0x5001D000ul)
CSL_CONTROLSS_G0_EPWM30_U_BASE	(0x5001E000ul)
CSL_CONTROLSS_G0_EPWM31_U_BASE	(0x5001F000ul)
CSL_CONTROLSS_G1_EPWM0_U_BASE	(0x50040000ul)
CSL_CONTROLSS_G1_EPWM1_U_BASE	(0x50041000ul)
CSL_CONTROLSS_G1_EPWM2_U_BASE	(0x50042000ul)
CSL_CONTROLSS_G1_EPWM3_U_BASE	(0x50043000ul)
CSL_CONTROLSS_G1_EPWM4_U_BASE	(0x50044000ul)
CSL_CONTROLSS_G1_EPWM5_U_BASE	(0x50045000ul)
CSL_CONTROLSS_G1_EPWM6_U_BASE	(0x50046000ul)
CSL_CONTROLSS_G1_EPWM7_U_BASE	(0x50047000ul)

CSL_CONTROLSS_G1_EPWM8_U_BASE	(0x50048000ul)
CSL_CONTROLSS_G1_EPWM9_U_BASE	(0x50049000ul)
CSL_CONTROLSS_G1_EPWM10_U_BASE	(0x5004A000ul)
CSL_CONTROLSS_G1_EPWM11_U_BASE	(0x5004B000ul)
CSL_CONTROLSS_G1_EPWM12_U_BASE	(0x5004C000ul)
CSL_CONTROLSS_G1_EPWM13_U_BASE	(0x5004D000ul)
CSL_CONTROLSS_G1_EPWM14_U_BASE	(0x5004E000ul)
CSL_CONTROLSS_G1_EPWM15_U_BASE	(0x5004F000ul)
CSL_CONTROLSS_G1_EPWM16_U_BASE	(0x50050000ul)
CSL_CONTROLSS_G1_EPWM17_U_BASE	(0x50051000ul)
CSL_CONTROLSS_G1_EPWM18_U_BASE	(0x50052000ul)
CSL_CONTROLSS_G1_EPWM19_U_BASE	(0x50053000ul)
CSL_CONTROLSS_G1_EPWM20_U_BASE	(0x50054000ul)
CSL_CONTROLSS_G1_EPWM21_U_BASE	(0x50055000ul)
CSL_CONTROLSS_G1_EPWM22_U_BASE	(0x50056000ul)
CSL_CONTROLSS_G1_EPWM23_U_BASE	(0x50057000ul)
CSL_CONTROLSS_G1_EPWM24_U_BASE	(0x50058000ul)
CSL_CONTROLSS_G1_EPWM25_U_BASE	(0x50059000ul)
CSL_CONTROLSS_G1_EPWM26_U_BASE	(0x5005A000ul)
CSL_CONTROLSS_G1_EPWM27_U_BASE	(0x5005B000ul)
CSL_CONTROLSS_G1_EPWM28_U_BASE	(0x5005C000ul)
CSL_CONTROLSS_G1_EPWM29_U_BASE	(0x5005D000ul)
CSL_CONTROLSS_G1_EPWM30_U_BASE	(0x5005E000ul)
CSL_CONTROLSS_G1_EPWM31_U_BASE	(0x5005F000ul)
CSL_CONTROLSS_G2_EPWM0_U_BASE	(0x50080000ul)
CSL_CONTROLSS_G2_EPWM1_U_BASE	(0x50081000ul)
CSL_CONTROLSS_G2_EPWM2_U_BASE	(0x50082000ul)
CSL_CONTROLSS_G2_EPWM3_U_BASE	(0x50083000ul)
CSL_CONTROLSS_G2_EPWM4_U_BASE	(0x50084000ul)
CSL_CONTROLSS_G2_EPWM5_U_BASE	(0x50085000ul)

CSL_CONTROLSS_G2_EPWM6_U_BASE	(0x50086000ul)
CSL_CONTROLSS_G2_EPWM7_U_BASE	(0x50087000ul)
CSL_CONTROLSS_G2_EPWM8_U_BASE	(0x50088000ul)
CSL_CONTROLSS_G2_EPWM9_U_BASE	(0x50089000ul)
CSL_CONTROLSS_G2_EPWM10_U_BASE	(0x5008A000ul)
CSL_CONTROLSS_G2_EPWM11_U_BASE	(0x5008B000ul)
CSL_CONTROLSS_G2_EPWM12_U_BASE	(0x5008C000ul)
CSL_CONTROLSS_G2_EPWM13_U_BASE	(0x5008D000ul)
CSL_CONTROLSS_G2_EPWM14_U_BASE	(0x5008E000ul)
CSL_CONTROLSS_G2_EPWM15_U_BASE	(0x5008F000ul)
CSL_CONTROLSS_G2_EPWM16_U_BASE	(0x50090000ul)
CSL_CONTROLSS_G2_EPWM17_U_BASE	(0x50091000ul)
CSL_CONTROLSS_G2_EPWM18_U_BASE	(0x50092000ul)
CSL_CONTROLSS_G2_EPWM19_U_BASE	(0x50093000ul)
CSL_CONTROLSS_G2_EPWM20_U_BASE	(0x50094000ul)
CSL_CONTROLSS_G2_EPWM21_U_BASE	(0x50095000ul)
CSL_CONTROLSS_G2_EPWM22_U_BASE	(0x50096000ul)
CSL_CONTROLSS_G2_EPWM23_U_BASE	(0x50097000ul)
CSL_CONTROLSS_G2_EPWM24_U_BASE	(0x50098000ul)
CSL_CONTROLSS_G2_EPWM25_U_BASE	(0x50099000ul)
CSL_CONTROLSS_G2_EPWM26_U_BASE	(0x5009A000ul)
CSL_CONTROLSS_G2_EPWM27_U_BASE	(0x5009B000ul)
CSL_CONTROLSS_G2_EPWM28_U_BASE	(0x5009C000ul)
CSL_CONTROLSS_G2_EPWM29_U_BASE	(0x5009D000ul)
CSL_CONTROLSS_G2_EPWM30_U_BASE	(0x5009E000ul)
CSL_CONTROLSS_G2_EPWM31_U_BASE	(0x5009F000ul)
CSL_CONTROLSS_G3_EPWM0_U_BASE	(0x500C0000ul)
CSL_CONTROLSS_G3_EPWM1_U_BASE	(0x500C1000ul)
CSL_CONTROLSS_G3_EPWM2_U_BASE	(0x500C2000ul)
CSL_CONTROLSS_G3_EPWM3_U_BASE	(0x500C3000ul)

CSL_CONTROLSS_G3_EPWM4_U_BASE	(0x500C4000ul)
CSL_CONTROLSS_G3_EPWM5_U_BASE	(0x500C5000ul)
CSL_CONTROLSS_G3_EPWM6_U_BASE	(0x500C6000ul)
CSL_CONTROLSS_G3_EPWM7_U_BASE	(0x500C7000ul)
CSL_CONTROLSS_G3_EPWM8_U_BASE	(0x500C8000ul)
CSL_CONTROLSS_G3_EPWM9_U_BASE	(0x500C9000ul)
CSL_CONTROLSS_G3_EPWM10_U_BASE	(0x500CA000ul)
CSL_CONTROLSS_G3_EPWM11_U_BASE	(0x500CB000ul)
CSL_CONTROLSS_G3_EPWM12_U_BASE	(0x500CC000ul)
CSL_CONTROLSS_G3_EPWM13_U_BASE	(0x500CD000ul)
CSL_CONTROLSS_G3_EPWM14_U_BASE	(0x500CE000ul)
CSL_CONTROLSS_G3_EPWM15_U_BASE	(0x500CF000ul)
CSL_CONTROLSS_G3_EPWM16_U_BASE	(0x500D0000ul)
CSL_CONTROLSS_G3_EPWM17_U_BASE	(0x500D1000ul)
CSL_CONTROLSS_G3_EPWM18_U_BASE	(0x500D2000ul)
CSL_CONTROLSS_G3_EPWM19_U_BASE	(0x500D3000ul)
CSL_CONTROLSS_G3_EPWM20_U_BASE	(0x500D4000ul)
CSL_CONTROLSS_G3_EPWM21_U_BASE	(0x500D5000ul)
CSL_CONTROLSS_G3_EPWM22_U_BASE	(0x500D6000ul)
CSL_CONTROLSS_G3_EPWM23_U_BASE	(0x500D7000ul)
CSL_CONTROLSS_G3_EPWM24_U_BASE	(0x500D8000ul)
CSL_CONTROLSS_G3_EPWM25_U_BASE	(0x500D9000ul)
CSL_CONTROLSS_G3_EPWM26_U_BASE	(0x500DA000ul)
CSL_CONTROLSS_G3_EPWM27_U_BASE	(0x500DB000ul)
CSL_CONTROLSS_G3_EPWM28_U_BASE	(0x500DC000ul)
CSL_CONTROLSS_G3_EPWM29_U_BASE	(0x500DD000ul)
CSL_CONTROLSS_G3_EPWM30_U_BASE	(0x500DE000ul)
CSL_CONTROLSS_G3_EPWM31_U_BASE	(0x500DF000ul)
CSL_CONTROLSS_ADC0_RESULT_U_BASE	(0x50100000ul)
CSL_CONTROLSS_ADC1_RESULT_U_BASE	(0x50101000ul)

CSL_CONTROLSS_ADC2_RESULT_U_BASE	(0x50102000ul)
CSL_CONTROLSS_ADC3_RESULT_U_BASE	(0x50103000ul)
CSL_CONTROLSS_ADC4_RESULT_U_BASE	(0x50104000ul)
CSL_CONTROLSS_CMPSSA0_U_BASE	(0x50200000ul)
CSL_CONTROLSS_CMPSSA1_U_BASE	(0x50201000ul)
CSL_CONTROLSS_CMPSSA2_U_BASE	(0x50202000ul)
CSL_CONTROLSS_CMPSSA3_U_BASE	(0x50203000ul)
CSL_CONTROLSS_CMPSSA4_U_BASE	(0x50204000ul)
CSL_CONTROLSS_CMPSSA5_U_BASE	(0x50205000ul)
CSL_CONTROLSS_CMPSSA6_U_BASE	(0x50206000ul)
CSL_CONTROLSS_CMPSSA7_U_BASE	(0x50207000ul)
CSL_CONTROLSS_CMPSSA8_U_BASE	(0x50208000ul)
CSL_CONTROLSS_CMPSSA9_U_BASE	(0x50209000ul)
CSL_CONTROLSS_CMPSSB0_U_BASE	(0x50220000ul)
CSL_CONTROLSS_CMPSSB1_U_BASE	(0x50221000ul)
CSL_CONTROLSS_CMPSSB2_U_BASE	(0x50222000ul)
CSL_CONTROLSS_CMPSSB3_U_BASE	(0x50223000ul)
CSL_CONTROLSS_CMPSSB4_U_BASE	(0x50224000ul)
CSL_CONTROLSS_CMPSSB5_U_BASE	(0x50225000ul)
CSL_CONTROLSS_CMPSSB6_U_BASE	(0x50226000ul)
CSL_CONTROLSS_CMPSSB7_U_BASE	(0x50227000ul)
CSL_CONTROLSS_CMPSSB8_U_BASE	(0x50228000ul)
CSL_CONTROLSS_CMPSSB9_U_BASE	(0x50229000ul)
CSL_CONTROLSS_ECAP0_U_BASE	(0x50240000ul)
CSL_CONTROLSS_ECAP1_U_BASE	(0x50241000ul)
CSL_CONTROLSS_ECAP2_U_BASE	(0x50242000ul)
CSL_CONTROLSS_ECAP3_U_BASE	(0x50243000ul)
CSL_CONTROLSS_ECAP4_U_BASE	(0x50244000ul)
CSL_CONTROLSS_ECAP5_U_BASE	(0x50245000ul)
CSL_CONTROLSS_ECAP6_U_BASE	(0x50246000ul)

CSL_CONTROLSS_ECAP7_U_BASE	(0x50247000ul)
CSL_CONTROLSS_ECAP8_U_BASE	(0x50248000ul)
CSL_CONTROLSS_ECAP9_U_BASE	(0x50249000ul)
CSL_CONTROLSS_DAC0_U_BASE	(0x50260000ul)
CSL_CONTROLSS_SDFM0_U_BASE	(0x50268000ul)
CSL_CONTROLSS_SDFM1_U_BASE	(0x50269000ul)
CSL_CONTROLSS_EQEP0_U_BASE	(0x50270000ul)
CSL_CONTROLSS_EQEP1_U_BASE	(0x50271000ul)
CSL_CONTROLSS_EQEP2_U_BASE	(0x50272000ul)
CSL_CONTROLSS_FSI_TX0_U_BASE	(0x50280000ul)
CSL_CONTROLSS_FSI_TX1_U_BASE	(0x50281000ul)
CSL_CONTROLSS_FSI_RX0_U_BASE	(0x50290000ul)
CSL_CONTROLSS_FSI_RX1_U_BASE	(0x50291000ul)
CSL_CONTROLSS_FSI_TX2_U_BASE	(0x502A0000ul)
CSL_CONTROLSS_FSI_TX3_U_BASE	(0x502A1000ul)
CSL_CONTROLSS_FSI_RX2_U_BASE	(0x502B0000ul)
CSL_CONTROLSS_FSI_RX3_U_BASE	(0x502B1000ul)
CSL_CONTROLSS_ADC0_U_BASE	(0x502C0000ul)
CSL_CONTROLSS_ADC1_U_BASE	(0x502C1000ul)
CSL_CONTROLSS_ADC2_U_BASE	(0x502C2000ul)
CSL_CONTROLSS_ADC3_U_BASE	(0x502C3000ul)
CSL_CONTROLSS_ADC4_U_BASE	(0x502C4000ul)
CSL_CONTROLSS_INPUTXBAR_U_BASE	(0x502D0000ul)
CSL_CONTROLSS_PWMXBAR_U_BASE	(0x502D1000ul)
CSL_CONTROLSS_PWMSYNCOUTXBAR_U_B ASE	(0x502D2000ul)
CSL_CONTROLSS_MDLXBAR_U_BASE	(0x502D3000ul)
CSL_CONTROLSS_ICLXBAR_U_BASE	(0x502D4000ul)
CSL_CONTROLSS_INTXBAR_U_BASE	(0x502D5000ul)
CSL_CONTROLSS_DMAXBAR_U_BASE	(0x502D6000ul)

CSL_CONTROLSS_OUTPUTXBAR_U_BASE	(0x502D8000ul)
CSL_CONTROLSS_OTTOCAL0_U_BASE	(0x502E0000ul)
CSL_CONTROLSS_OTTOCAL1_U_BASE	(0x502E1000ul)
CSL_CONTROLSS_OTTOCAL2_U_BASE	(0x502E2000ul)
CSL_CONTROLSS_OTTOCAL3_U_BASE	(0x502E3000ul)
CSL_CONTROLSS_CTRL_U_BASE	(0x502F0000ul)
CSL_DEBUGSS_U_BASE	(0x50800000ul)
CSL_MSS_CTRL_U_BASE	(0x50D00000ul)
CSL_TOP_CTRL_U_BASE	(0x50D80000ul)
CSL_SPINLOCK0_BASE	(0x50E00000ul)
CSL_VIM_U_BASE	(0x50F00000ul)
CSL_GPIO0_U_BASE	(0x52000000ul)
CSL_GPIO1_U_BASE	(0x52001000ul)
CSL_GPIO2_U_BASE	(0x52002000ul)
CSL_GPIO3_U_BASE	(0x52003000ul)
CSL_WDT0_U_BASE	(0x52100000ul)
CSL_WDT1_U_BASE	(0x52101000ul)
CSL_WDT2_U_BASE	(0x52102000ul)
CSL_WDT3_U_BASE	(0x52103000ul)
CSL_RTI0_U_BASE	(0x52180000ul)
CSL_RTI1_U_BASE	(0x52181000ul)
CSL_RTI2_U_BASE	(0x52182000ul)
CSL_RTI3_U_BASE	(0x52183000ul)
CSL_MCSPI0_U_BASE	(0x52200000ul)
CSL_MCSPI1_U_BASE	(0x52201000ul)
CSL_MCSPI2_U_BASE	(0x52202000ul)
CSL_MCSPI3_U_BASE	(0x52203000ul)
CSL_MCSPI4_U_BASE	(0x52204000ul)
CSL_UART0_U_BASE	(0x52300000ul)
CSL_UART1_U_BASE	(0x52301000ul)

CSL_UART2_U_BASE	(0x52302000ul)
CSL_UART3_U_BASE	(0x52303000ul)
CSL_UART4_U_BASE	(0x52304000ul)
CSL_UART5_U_BASE	(0x52305000ul)
CSL_LIN0_U_BASE	(0x52400000ul)
CSL_LIN1_U_BASE	(0x52401000ul)
CSL_LIN2_U_BASE	(0x52402000ul)
CSL_LIN3_U_BASE	(0x52403000ul)
CSL_LIN4_U_BASE	(0x52404000ul)
CSL_I2C0_U_BASE	(0x52500000ul)
CSL_I2C1_U_BASE	(0x52501000ul)
CSL_I2C2_U_BASE	(0x52502000ul)
CSL_I2C3_U_BASE	(0x52503000ul)
CSL_MCAN0_MSG_RAM_U_BASE	(0x52600000ul)
CSL_MCAN0_CFG_U_BASE	(0x52608000ul)
CSL_MCAN1_MSG_RAM_U_BASE	(0x52610000ul)
CSL_MCAN1_CFG_U_BASE	(0x52618000ul)
CSL_MCAN2_MSG_RAM_U_BASE	(0x52620000ul)
CSL_MCAN2_CFG_U_BASE	(0x52628000ul)
CSL_MCAN3_MSG_RAM_U_BASE	(0x52630000ul)
CSL_MCAN3_CFG_U_BASE	(0x52638000ul)
CSL_MCAN0_ECC_U_BASE	(0x52700000ul)
CSL_MCAN1_ECC_U_BASE	(0x52701000ul)
CSL_MCAN2_ECC_U_BASE	(0x52702000ul)
CSL_MCAN3_ECC_U_BASE	(0x52703000ul)
CSL_CPSW0_U_BASE	(0x52800000ul)
CSL_TPCC0_U_BASE	(0x52A00000ul)
CSL_TPTC00_U_BASE	(0x52A40000ul)
CSL_TPTC01_U_BASE	(0x52A60000ul)
CSL_DCC0_U_BASE	(0x52B00000ul)

CSL_DCC1_U_BASE	(0x52B01000ul)
CSL_DCC2_U_BASE	(0x52B02000ul)
CSL_DCC3_U_BASE	(0x52B03000ul)
CSL_TOP_ESM_U_BASE	(0x52D00000ul)
CSL_SOC_TIMESYNC_XBAR0_U_BASE	(0x52E00000ul)
CSL_EDMA_TRIG_XBAR_U_BASE	(0x52E01000ul)
CSL_GPIO_INTR_XBAR_U_BASE	(0x52E02000ul)
CSL_ICSSM_INTR_XBAR_U_BASE	(0x52E03000ul)
CSL_SOC_TIMESYNC_XBAR1_U_BASE	(0x52E04000ul)
CSL_ECC_AGG_R5SS0_CORE0_U_BASE	(0x53000000ul)
CSL_ECC_AGG_R5SS0_CORE1_U_BASE	(0x53003000ul)
CSL_ECC_AGG_R5SS1_CORE0_U_BASE	(0x53004000ul)
CSL_ECC_AGG_R5SS1_CORE1_U_BASE	(0x53007000ul)
CSL_ECC_AGG_TOP_U_BASE	(0x53010000ul)
CSL_IOMUX_U_BASE	(0x53100000ul)
CSL_TOP_RCM_U_BASE	(0x53200000ul)
CSL_MSS_RCM_U_BASE	(0x53208000ul)
CSL_R5SS0_CCMR_U_BASE	(0x53210000ul)
CSL_R5SS1_CCMR_U_BASE	(0x53211000ul)
CSL_TOP_PBIST_U_BASE	(0x53300000ul)
CSL_R5SS0_STC_U_BASE	(0x53500000ul)
CSL_R5SS1_STC_U_BASE	(0x53510000ul)
CSL_TOP_EFUSE_FARM_U_BASE	(0x53600000ul)
CSL_EXT_FLASH0_U_BASE	(0x60000000ul)
CSL_EXT_FLASH1_U_BASE	(0x62000000ul)
CSL_GPMC0_MEM_U_BASE	(0x68000000ul)
CSL_L2OCRAM_U_BASE	(0x70000000ul)
CSL_MBOX_SRAM_U_BASE	(0x72000000ul)
CSL_R5SS0_CORE0_ICACHE_U_BASE	(0x74000000ul)
CSL_R5SS0_CORE0_DCACHE_U_BASE	(0x74800000ul)

CSL_R5SS0_CORE1_ICACHE_U_BASE	(0x75000000ul)
CSL_R5SS0_CORE1_DCACHE_U_BASE	(0x75800000ul)
CSL_R5SS1_CORE0_ICACHE_U_BASE	(0x76000000ul)
CSL_R5SS1_CORE0_DCACHE_U_BASE	(0x76800000ul)
CSL_R5SS1_CORE1_ICACHE_U_BASE	(0x77000000ul)
CSL_R5SS1_CORE1_DCACHE_U_BASE	(0x77800000ul)
CSL_R5SS0_CORE0_TCMA_U_BASE	(0x78000000ul)
CSL_R5SS0_CORE0_TCMB_U_BASE	(0x78100000ul)
CSL_R5SS0_CORE1_TCMA_U_BASE	(0x78200000ul)
CSL_R5SS0_CORE1_TCMB_U_BASE	(0x78300000ul)
CSL_R5SS1_CORE0_TCMA_U_BASE	(0x78400000ul)
CSL_R5SS1_CORE0_TCMB_U_BASE	(0x78500000ul)
CSL_R5SS1_CORE1_TCMA_U_BASE	(0x78600000ul)
CSL_R5SS1_CORE1_TCMB_U_BASE	(0x78700000ul)
CSL_HSM_DTHE_U_BASE	(0xCE000000ul)
CSL_HSM_SHA_U_BASE	(0xCE004000ul)
CSL_HSM_AES_U_BASE	(0xCE006000ul)
CSL_HSM_TRNG_U_BASE	(0xCE00A000ul)
CSL_HSM_PKA_U_BASE	(0xCE010000ul)
CSL_HSM_PKA_RAM_U_BASE	(0xCE014000ul)

9.2. Parametri izmjenjivača

```
#ifndef TRINV_PARAM_H_
#define TRINV_PARAM_H_

//
// Define the system frequency (MHz)
//
#define CONTROLSS_FREQUENCY    200.0
#define SYSTEM_FREQUENCY       CONTROLSS_FREQUENCY

//
// PWM, SAMPLING FREQUENCY and Current Loop Band width definitions (KHz)
//
#define PWM_FREQUENCY           10.0

//
// This line sets the SAMPLING FREQUENCY to one of the available choices
//
#define  SINGLE_SAMPLING        1
#define  DOUBLE_SAMPLING        2

//
// User can select choices from available control configurations
//
#define  SAMPLING_METHOD     SINGLE_SAMPLING
//#define  SAMPLING_METHOD     DOUBLE_SAMPLING

#if(SAMPLING_METHOD == SINGLE_SAMPLING)
#define ISR_FREQUENCY           (PWM_FREQUENCY)
#define ISR_RES_RATIO           0U

#elif(SAMPLING_METHOD == DOUBLE_SAMPLING)
#define ISR_FREQUENCY           (2.0 * PWM_FREQUENCY)
#define ISR_RES_RATIO           1U

#endif

//
// Keep PWM Period same between single sampling and double sampling
//
// Change to SYSTEM_FREQUENCY / 1 after changing EPWM Clock Divide Select to /1
//
#define INV_PWM_TICKS        ((SYSTEM_FREQUENCY / PWM_FREQUENCY) * 1000.0)
#define INV_PWM_DB            (200.0)
#define QEP_UNIT_TIMER_TICKS  \
    (SYSTEM_FREQUENCY / (2.0 * PWM_FREQUENCY) * 1000.0)

#define INV_PWM_TBPRD         (INV_PWM_TICKS / 2.0)
#define INV_PWM_HALF_TBPRD    (INV_PWM_TBPRD / 2.0)
#define SAMPLING_FREQ         (ISR_FREQUENCY * 1000)
#define CUR_LOOP_BANDWIDTH    (2.0 * PI * SAMPLING_FREQ / 18.0)

#define TPWM_CARRIER          (1000.0 / PWM_FREQUENCY)

//
// Define the base quantites
//
#define BASE_VOLTAGE          340.0f  // Base peak phase voltage (volt)
#define BASE_CURRENT          10.0f  // Base peak phase current (amp)
#define BASE_TORQUE                   // Base torque (N.m)
#define BASE_FLUX                     // Base flux linkage (volt.sec/rad)
```

```
#define BASE_FREQ             50.0f  // Base electrical frequency (Hz)

#define MOTOR_TEMP_LIMIT      90.0f

#define CURRENT_LIMIT         1.5f

#define VDC_FLT_BW_HZ       2.0f

#define VQ_LIMIT            1.15f
#define MODULATION_LIMIT    1.33f

//
// Current sensors scaling
//
#define AMC1302(A)      (2048 * A / BASE_CURRENT)
#define AMC1311(A)      (4096 * A / BASE_VOLTAGE)

//
// Analog scaling with ADC
//
// 1/2^12
//
#define ADC_PU_SCALE_FACTOR        0.000244140625f

// Skaliranje adc ocitanja napona za referentni napon i djelitelj napona
// (1/broj razina adc) * (referentni napon) * (djelitelj napona)
#define ADC_V_REFRENCE_SCALE       (ADC_PU_SCALE_FACTOR) * (3.22693f) * (125.0f)



//
// 1/2^11
//
#define ADC_PU_PPB_SCALE_FACTOR    0.000488281250f
#define ADC_I_SCALE_FCT 0.0012605195f

#define SD_VOLTAGE_SENSE_SCALE     (SD_PU_SCALE_FACTOR * (100.0f / 0.212f))

//
// Constants for ADC sampling delay calculation
//
// Sample and hold time
//
#define ADC_S_H_TIME_NS     75.0f
//
// Conversion time
//
#define ADC_CONV_TIME_NS    175.0f
#define NS_TO_S             1.0e-9f

//
// ADC Related defines
//
#define R_REF            ADC_readResult   (CSL_CONTROLSS_ADC0_RESULT_U_BASE,
ADC_SOC_NUMBER1)

#define R_EXC            ADC_readResult   (CSL_CONTROLSS_ADC4_RESULT_U_BASE,
ADC_SOC_NUMBER0)
//#define R_EXC_PPB        ADC_readPPBResult(CONFIG_ADC4, ADC_PPB_NUMBER1)

#define R_SIN1       ADC_readResult(CSL_CONTROLSS_ADC4_RESULT_U_BASE,
ADC_SOC_NUMBER0)
```

```
#define R_SIN2        ADC_readResult(CSL_CONTROLSS_ADC4_RESULT_U_BASE,
ADC_SOC_NUMBER3)
//#define R_SIN_PPB   ADC_readPPBResult(CONFIG_ADC4, ADC_PPB_NUMBER1)

#define R_COS1        ADC_readResult(CSL_CONTROLSS_ADC4_RESULT_U_BASE,
ADC_SOC_NUMBER1)
#define R_COS2        ADC_readResult(CSL_CONTROLSS_ADC4_RESULT_U_BASE,
ADC_SOC_NUMBER1)
//#define R_COS_PPB   ADC_readPPBResult(CONFIG_ADC4, ADC_PPB_NUMBER2)

#define IFBU         ADC_readResult(CSL_CONTROLSS_ADC1_RESULT_U_BASE,
ADC_SOC_NUMBER0)
#define IFBV         ADC_readResult(CSL_CONTROLSS_ADC2_RESULT_U_BASE,
ADC_SOC_NUMBER0)
// IFBW nije struja faze W, vec struja zvijezdista (Ifb-ret)
#define IFBW         ADC_readResult(CSL_CONTROLSS_ADC3_RESULT_U_BASE,
ADC_SOC_NUMBER0)

#define IFBU_PPB     ADC_readPPBResult(CSL_CONTROLSS_ADC1_RESULT_U_BASE,
ADC_PPB_NUMBER1)
#define IFBV_PPB     ADC_readPPBResult(CSL_CONTROLSS_ADC2_RESULT_U_BASE,
ADC_PPB_NUMBER1)
// IFBW nije struja faze W, vec struja zvijezdista (Ifb-ret)
#define IFBW_PPB     ADC_readPPBResult(CSL_CONTROLSS_ADC3_RESULT_U_BASE,
ADC_PPB_NUMBER1)

#define VDC_EVT      ADC_readResult(CSL_CONTROLSS_ADC0_RESULT_U_BASE,
ADC_SOC_NUMBER0)

#define TEMP_SENS    ADC_readResult(CSL_CONTROLSS_ADC1_RESULT_U_BASE,
ADC_SOC_NUMBER3)

#endif /* TRINV_PARAM_H_ */
```

9.3. Međuprocesorska komunikacija

```
#include "IPC_RPC_Comm.h"
#include "FOC_loop.h"
#include "Encoder.h"
// #include "foc.h"
#include <drivers/ipc_rpmsg.h>

#define RECEVE_ENDPOINT 10
#define TCP_STREAM_ENDPOINT 12
#define TCP_SERVER_CORE_ID CSL_CORE_ID_R5FSS0_1


typedef struct __CommandPacket {
   int endpoint;
   char value[64];
} CommandPacket;


RPMessage_Object gRecvMsgObject;
void IPC_SendMessage(void *data, uint16_t dataLen);
void IPC_SendInverterDataPacket();
unsigned short SendNewIPCRPPacket;
uint32_t packetTime;


void setup_IPC()
{
    RPMessage_CreateParams createParams;
    RPMessage_CreateParams_init(&createParams);
    createParams.localEndPt = RECEVE_ENDPOINT;
    RPMessage_construct(&gRecvMsgObject, &createParams);
    SendNewIPCRPPacket = 0;
    packetTime = 0;
}

void processIPC()
{
    if (SendIPCRPPacket == 1) {
        // Reset the flag
        SendIPCRPPacket = 0;
        IPC_SendDataPacket();
    }

    CommandPacket commandPacket = { -1, 0 };
    char receveBuffer[sizeof(commandPacket)] = {0};
    uint16_t replyMsgSize = sizeof(receveBuffer);
    uint16_t remoteCoreId, remoteCoreEndPt;

    int32_t status = RPMessage_recv(
        &gRecvMsgObject,
        receveBuffer,
        &replyMsgSize,
        &remoteCoreId,
        &remoteCoreEndPt,
        1);

    memcpy(&(commandPacket.endpoint), receveBuffer,
sizeof(commandPacket.endpoint));
```

```
    memcpy(&(commandPacket.value), receveBuffer + sizeof(commandPacket.endpoint),
sizeof(commandPacket.value));

    if ( commandPacket.endpoint == -1) {
        return;
    }

    if ( commandPacket.endpoint == 1) {
        if (strcmp((char *)commandPacket.value, "ON") == 0) {
            FOC_setMotorRunState(1);
        }
        else {
            FOC_setMotorRunState(0);
    }
    else if ( commandPacket.endpoint == 2) {
        FOC_setIdref(atof(commandPacket.value));
    }
    else if ( commandPacket.endpoint == 3) {
        FOC_setSpeedRef(atof(commandPacket.value));
    }
}


void IPC_SendMessage(void *data, uint16_t dataLen)
{
    RPMessage_send(
        data,
        dataLen,
        TCP_SERVER_CORE_ID,
        TCP_STREAM_ENDPOINT,
        RPMessage_getLocalEndPt(&gRecvMsgObject),
        0);
}
```

9.4. Glavna metoda logike izmjenjivača

```
#include "device.h"

#include "Serial_Cmd_HAL.h"
#include "Serial_CLI.h"
#include "ucc5870.h"

#include "FOC_loop.h"
#include "IPC_RPC_Comm.h"


/* Global variables and objects */
Bool gFlag = TRUE;
uint16_t gLoopTicker = 0;


void trinv_init(void)
{
    //
    // Select resolver sin/cos swap mux
    //
    uint32_t    gpioBaseAddr, pinNum;
    gpioBaseAddr = (uint32_t)
AddrTranslateP_getLocalAddr(CONFIG_GPIO_RES_INH_BASE_ADDR);
    pinNum       = CONFIG_GPIO_RES_INH_PIN;
    GPIO_setDirMode(gpioBaseAddr, pinNum, CONFIG_GPIO_RES_INH_DIR);
    GPIO_pinWriteHigh(gpioBaseAddr, pinNum);

}


void trinv_main(void *args)
{
    DebugP_log("================================\r\n");
    DebugP_log("  Ritroller ::  Motor Controller \r\n");
    DebugP_log("================================\r\n");
    /* Open drivers to open the UART driver for console */
    Drivers_open();
    Board_driversOpen();
    trinv_init();
    FOC_init();
    FOC_run();
    IpcNotify_syncAll(SystemP_WAIT_FOREVER);
    setup_IPC();
    SerialCmd_init();
    serial_cli_init();
    while (gFlag){

        SerialCmd_read();
        serial_cli_service();
        processIPC();

    }
    Board_driversClose();
    Drivers_close();
}
```