UNIVERZA V LJUBLJANI
FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Sašo Vrbinc

Servisni program za lokalno prikazovalno enoto zaščitnega releja

DIPLOMSKO DELO
NA VISOKOŠOLSKEM STROKOVNEM ŠTUDIJU

Mentor: doc. dr. Boštjan Slivnik

Ljubljana, 2009

Rezultati diplomskega dela so intelektualna lastnina Fakultete za računalništvo in informatiko Univerze v Ljubljani. Za objavljanje ali izkoriščanje rezultatov diplomskega dela je potrebno pisno soglasje Fakultete za računalništvo in informatiko ter mentorja.

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Namesto te strani **vstavite** original izdane teme diplomskega dela s podpisom mentorja in dekana ter žigom fakultete, ki ga diplomant dvigne v študentskem referatu, preden odda izdelek v vezavo!

IZJAVA O AVTORSTVU

diplomskega dela

Spodaj podpisani Sašo Vrbinc,

z vpisno številko 63030082,

sem avtor diplomskega dela z naslovom:

Servisni program za lokalno prikazovalno enoto zaščitnega releja

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom doc. dr. Boštjana Slivnika
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela
- soglašam z javno objavo elektronske oblike diplomskega dela v zbirki "Dela FRI".

V Ljubljani, dne 1.6.2009

Zahvala

Iskreno se zahvaljujem vsem tistim, ki so kakorkoli pomagali pri nastanku mojega diplomskega dela. Posebno zahvalo namenjam mentorju doc. dr. Boštjanu Slivniku za vse koristne nasvete in korekten strokovni odnos, Iztoku Kobalu in Gregi Flandru za strokovno pomoč in pojasnila na problemski domeni, sodelavcem in prijateljem, mojemu očetu, bratu ter Suzani, ki so me bodrili in verjeli vame ter mi pomagali na trnovi poti do uspeha.

Posvečeno očetu, bratu in Suzani

Kazalo

Povzetek **1**

1 Uvod **3**
1.1 Zaščita elektroenergetskih sistemov . . . 3
1.2 Sekundarni elektroenergetski sistem . . . 4
1.3 Funkcionalnost LDU . . . 5
1.4 Cilji . . . 6

2 Protokol Modbus **7**
2.1 Struktura telegramov . . . 7
2.2 Funkcije . . . 8
2.2.1 Funkcija 4 . . . 9
2.2.2 Funkcija 6 . . . 10
2.3 Modifikacija protokola . . . 10
2.4 Arhitektura . . . 12
2.5 Izvedba protokola . . . 13
2.5.1 Sprejem, oddaja in dekodiranje telegramov . . . 14
2.6 Testiranje . . . 15
2.7 Pomankljivosti in izboljšave . . . 17

3 Funkcionalnosti lokalne prikazovalne enote **18**
3.1 Načrtovanje in izvedba zasnove . . . 18
3.1.1 Manipulacija registrske baze . . . 19
3.1.2 Vzorec singleton in večpoljska podpora . . . 19
3.2 Viri podatkov . . . 21
3.2.1 XML . . . 22
3.2.2 Atomična baza . . . 22
3.3 Serializacija . . . 23
3.4 Osnovna stran . . . 23

3.4.1 Slepa (neaktivna) shema polja 24
3.4.2 Animirani elementi 26
3.4.3 Statusi animiranih elementov 27
3.4.4 Ukazi 29
3.5 Preklop vodenja 31
3.6 Analogne meritve 32
3.7 Statusi digitalnih vhodov 35
3.8 Dogodki in alarmi 36
3.8.1 Predelava načina sprejema dogodkov 43
3.9 Uporabniško nastavljive diode LED 45

4 Sklepne ugotovitve **47**

A Sekvenčna diagrama komunikacije **49**

Seznam slik **49**
Seznam tabel **53**

Literatura **54**

Seznam uporabljenih kratic in simbolov

HMI Human-Machine Interface, včasih MMI (Man-Machine Interface) ali GUI (Graphical User Interface), označuje široko uporabljen izraz, ki služi kot sopomenka uporabniškemu vmesniku

LED Polprevodniška dioda, ki emitira svetlobo

EES Okrajšava za elektroenergetski sistem

XML Oznaka za označevalni jezik (Extensible Markup Language)

UML Standardizirani jezik za modeliranje na področju razvoja programske opreme, ki vključuje množico grafičnih orodij

IKS Informacijsko Komunikacijski Sistem

FIFO Princip obvladovanje čakajočih podatkov v vrsti, kjer se najprej obdelajo najstarejši (First In - First Out)

OOP Oznaka za objektno usmerjeno programiranje (Object Orientated Programming)

STL Knjižnica, ki vsebuje pogosto uporabljene dinamične kontejnerje, iteratorje, algoritme in funktorje (Standard Template Library)

MDB Multi-drop vodilo

RTU Remote Terminal Unit

Povzetek

Uporabniški vmesniki predstavljajo nepogrešljiv člen pri upravljanju z elektronskimi napravami in zaščitni releji v elektroenergetiki niso izjema. Pripadajoči del zaščitnih relejev so tudi lokalne prikazovalne enote, katerih namen je prikazovanje stanj in upravljanje primarnega elektroenergetskega sistema (EES). Funkcionalnost lokalne prikazovalne enote tvorita periferni prikazovalnik (gospodar) in servisni program (suženj) na kontrolni enoti zaščitnega releja, ki komunicirata po mrežnem vmesniku RS-232. Cilj naloge je ustvariti servisni program na obstoječi arhitekturi informacijsko komunikacijskega sistema (IKS) rcman, ki je sposoben komunicirati z gospodarjem s standardiziranim protokolom Modbus ter zagotoviti serviranje zahtev gospodarja iz množice podprtih funkcionalnosti. V protokol sem implementiral nabor standardnih funkcij, ki so zadoščale potrebam po zadostitvi zahtev gospodarja. V začetni fazi razvoja se je pokazala pomanjkljivost protokola, zato sem naredil strukturno modifikacijo in posledično izgubil kompatibilnost s standardom. Ključni del servisnega programa je zahteval učinkovito serviranje podatkov in enostavno dodajanje novih funkcionalnosti, pri čemer sem izkoristil zmožnosti objektno usmerjenega programskega jezika C++. Rezultat dela je delujoč servisni program, ki izvaja strežbo podatkov alarmov, dogodkov, analognih meritev, indikacij LED ter stanja digitalnih vhodov.

Ključne besede:

servisni program, protokol Modbus, zaščitni rele

Abstract

Local display unit functions as a user interface of the protection relay. It is therefore an indispensable part of the relay, used for monitoring and controlling of the relay and,consequently, of certain power grid elements. Local display unit is composed of two parts: a standalone peripheral display unit (the master device), and a software driver (the slave) running on the protection relay control unit. The thesis describes a software driver which is incorporated into the existing architecture of the 'rcman' information-communication system, and runs on the protection relay. The software driver communicates with the master device using the Modbus protocol over an RS-232 interface. The two key requirements to be met in preparation of the thesis were a high data throughput, on one hand, and a simple extensibility of the driver on the other hand. During the initial development phase it was established that the implemented subset of Modbus functions is not sufficient for communication between the driver and the peripheral display unit. Therefore,certain Modbus functions had to be modified at the expense of losing their compatibility with the standard Modbus protocol. The result is a working software driver that meets the above mentioned requirements and provides the peripheral display unit with the data describing alarms, events, analogue measurements,LED indications, and the status of digital input lines.

Key words:

software driver, protocol Modbus, protection relay

Poglavje 1

Uvod

1.1 Zaščita elektroenergetskih sistemov

Elektroenergetski sistemi (EES) so ustvarjeni z namenom, da zagotovijo neprekinjeno generiranje, transformacijo in porabo električne energije. V sistemu prihaja do neizogibljivih nihanj, ki jih je potrebno spremljati, saj lahko preidejo v območje izven tolerance in povzročijo poškodbe ali uničenje sosednjih komponent.

Možnost abnormalnega stanja, v katerega lahko preide sistem je majhna, vendar se dogaja. Možni povzročitelji za takšna stanja so strele, izredni vremenski pogoji ali zastarela oprema. Okvare so pogostejše v primeru, če ni opravljenih ustreznih vzdrževalnih del, zato se z redno menjavo dotrajanih naprav z novejšimi in kvalitetnejšimi, zmanjša njihovo pogostost.

Energetske sisteme členimo na primarne in sekundarne. Primarni sistem tvorijo

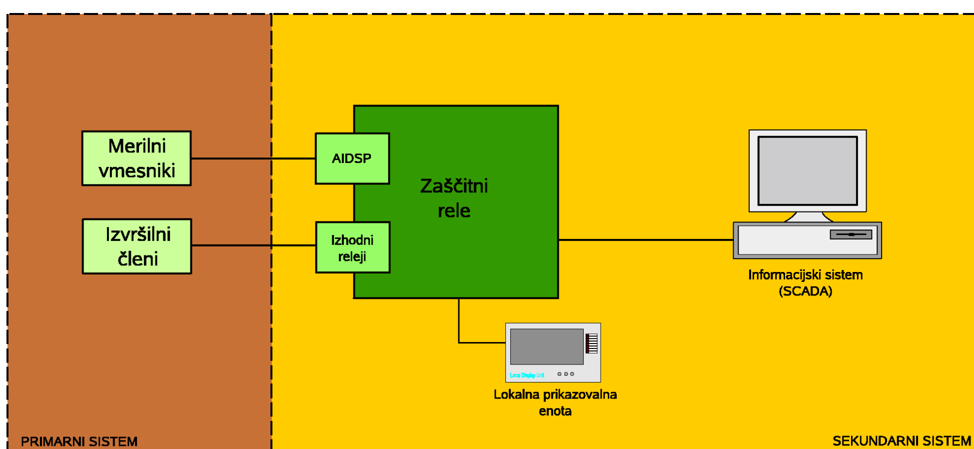
- **transformatorji**,
- **generatorji**,
- **zbiralke** in
- **vodi**,

ki delujejo v dovoljenem tokovnem, napetostnem, električnem in energijskem območju. Komponente so med seboj povezane v raznih kombinacijah z uporabo ločilk in odklopnikov. Odklopniki so zmožni prekiniti električni tok pri visokih napetostih v primeru okvare na posamezni komponenti. Za mehanizem kontrole preklopa je odgovoren **zaščitni rele**, ki sodi v del sekundarnega EES.

Skupina komponent povezanih z generiranjem, preklapljanjem, transformacijo ali porabo, so elektrarne in podpostaje. Zaščitna, zajemalna, kontrolna in komunikacijska oprema je nameščena na vsaki izmed omenjenih kompleksov.

1.2 Sekundarni elektroenergetski sistem

Glavna naloga sekundarnega sistema je njegova zaščitna funkcija, ki jo po navadi spremljajo funkcije zajema in kontrole. Za opravljanje teh funkcij je na vsako komponento primarnega sistema nameščen zaščitni rele. Komponente sekundarnega sistema so prikazane na sliki 1.1.



Slika 1.1: Komponente sekundarnega elektroenergetskega sistema v relaciji s primarnim

- *Merilni vmesniki* so v osnovi merilni transformatorji za tokovne in napetostne meritve. Predstavljajo vmesnik med elementi primarnega sistema in zaščitnega releja.

- *AIDSP* enota je del zaščitnega releja, ki skrbi za pretvarjanje vhodnih analognih signalov v digitalne.

- *Zaščitni rele* ima poleg AIDSP in izhodnih relejev še CPU in druge procesorje za izvajanje časovno kritičnih operacij. V primeru presežka meritve iz območja tolerance, sistem na osnovi pridobljenih meritev pošlje ustreznemu releju signal za preklop izvršilnega člena.

- *Izvršilni členi* so navadno odklopniki ali ločilke, ki prekinejo tok tedaj, ko prispe signal iz izhodnega releja.

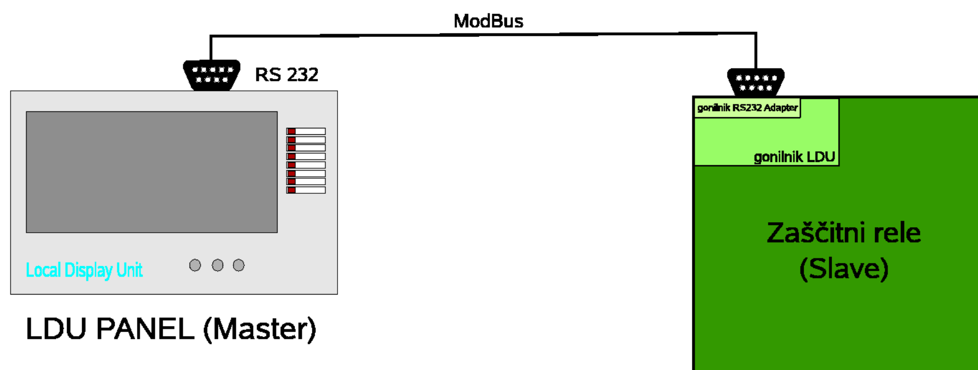
- *Informacijski sistem* je sistem za daljinsko kontrolo vodenja in koncentriranja podatkov zaščitnih relejev. Vsi pridobljeni podatki se hranijo v relacijsko bazo in so s strani inženirjev skrbno nadzorovani prek uporabniških vmesnikov.

V sodobnih zaščitnih relejih so pogosto nameščene dodatne komponente za povečanje varnosti, nadzora in učinkovitosti. To so razni prikazovalniki in indikatorji oziroma lokalne prikazovalne enote (LDU), ki nudijo zaščitnemu inženirju ažurne podatke o stanju elementov primarnega sistema na lokalnem nivoju.

1.3 Funkcionalnost LDU

Preden lahko predstavim namen in cilj diplomske naloge, sem primoran napisati nekaj o komponentah celotne funkcionalnosti LDU. Razloga za to sta predvsem obseg in delitev nalog z drugimi razvijalci. V diplomski nalogi opisujem komunikacijski protokol Modbus [6] in gonilnik LDU ter delno gonilnik za sprejemanje dogodkov, saj je bilo načrtovanje in implementacija izvedena z moje strani.

Funkcionalnost LDU tvorita periferna prikazovalna enota in servisni program na zaščitnem releju, ki komunicirata po vmesniku RS-232. Delujeta po principu gospodar-suženj, kjer je prikazovalna enota gospodar in zaščitni rele suženj. Periferna enota ima svoj mikrokontroler, na katerega je vezan prikazovalnik, nabor funkcijskih tipk in indikatorji LED.



Slika 1.2: Shema funkcionalnosti LDU

Za uspešno medsebojno komuniciranje deluje ustrezen komunikacijski protokol Modbus, ki je de-facto standard na omrežjih RS-232 in Ethernet.

Gonilniki na zaščitnem releju so izvedeni po vzoru drugih, bolj znanih večplastnih omrežnih protokolov kot so ISO/OSI ali TCP/IP. Temu sledi tudi servisni program s protokolom Modbus, ki sem ga implementiral kot modul na aplikacijski arhitekture rcman.

Glede na izvedenost gonilnika je njegov pripadajoči del še funkcionalnost za sprejem dogodkov imenovan EventDriver, brez katerega ni možno oziroma je močno oteženo realizirati določenih funkcionalnosti v servisnem programu.

1.4 Cilji

Pred začetkom načrtovanja in implementacije servisnega programa kot modula obstoječe arhitekture rcman, ki ga razlagam v razdelku 2.5, sem implementiral protokol Modbus za komunikacijo s prikazovalno enoto, s katero sem zagotovil izmenjavo sporočil. Sprva sem se striktno držal standardizacije, vendar so se kmalu izkazale njegove pomanjkljivosti, zato se nisem mogel izogniti določenim strukturnim modifikacijam.

Najobsežnejša in najtežja naloga v diplomskem delu, ki se je razprostirala skozi tri razvojne iteracije, je implementacija serviranja podatkov posameznih funkcionalnosti. V vsaki iteraciji mi je bila dodeljena določena množica funkcionalnosti, ki jih v diplomskem delu obravnavam kot celoto in ne delam vidne distinkcije med posameznimi fazami. Glede na zahtevnik sem v gonilnik omogočil posredovanje informacij o naslednjih funkcionalnostih: dogodki in alarmi, analogne meritve, stanja digitalnih vhodov ter druge funkcionalnosti, ki služijo za kasnejši prikaz in indikacijo relevantnih informacij zaščitnemu inženirju na prikazovalni enoti. Implementacija servisnega programa je zahtevala izvedbo v objektno usmerjenem programskem jeziku C++.

Določene funkcionalnosti servisnega programa, so odvisne od gonilnika za hranjenje dogodkov imenovanega EventDriver. Obstoječega sem integriral v servisni program in zaradi določenih težav tudi predelal sistem za sprejemanje dogodkov.

Poglavje 2

Protokol Modbus

Razvoj standardnega protokola Modbus sega v leto 1979, osnovni namen protokola pa je bila komunikacija na multi-drop vodilih (MDB) z komunikacijskim modelom gospodar-suženj (angl. master/slave). Sprva je bil namenjen komunikaciji na omrežnem vmesniku RS-232, kasneje pa se je razširil tudi na ostale, hitrejše vmesnike (RS485, Ethernet). Zaradi velike razširjenosti v industriji je kmalu postal de-facto standard in eden izmed najbolj razširjenih komunikacijskih protokolov na področju elektronskih naprav.

Kot vsi ostali gonilniki v arhitekturi rcman je tudi Modbus implementiran kot pripadajoči modul, zato sem moral najprej proučiti vmesnike, ki jih narekuje omenjen sistem. Šele nato sem lahko začel z implementacijo protokola. Ta vključuje dve standardizirani funkciji, ki sta zadoščali za realizacijo funkcionalnosti. Po naknadno ugotovljeni slabosti protokola sem bil primoran modificirati strukturo telegrama, kar je posledično pomenilo izgubo kompatibilnosti s standardom.

2.1 Struktura telegramov

Komunikacija med posameznimi vozlišči v omrežju poteka s sporočili v obliki telegramov in so neodvisni od fizičnega vmesnika po katerem se prenašajo. Posledična prednost je nepotrebna reimplementacija protokola za vsak fizični vmesnik posebej. V našem primeru je edini tip vmesnika RS-232, po katerem poteka komunikacija med prikazovalno enoto in zaščitnim relejem.

Komunikacija med vozlišči poteka v binarnem načinu (RTU). Standard opredeljuje tudi ASCII način komunikacije, ki pa ga implementacija ne vključuje.

Struktura telegrama je odvisna od funkcije ter izvora pošiljanja in je v splošnem v obliki kot jo prikazuje tabela 2.1.

Glava	Podatki	Kontrola napake

Tabela 2.1: Splošna struktura telegrama

Struktura glave vsebuje naslednje informacije:

- *Naslov naprave* - Identifikacijska oznaka naprave sužnja, ki ima podobno vlogo kot številka IP v protokolu TCP/IP. Velika pomanjkljivost se izkaže na širšem omrežju z veliko napravami, saj jih več kot 247 ne more biti vključenih. Naslov ne sme biti 0, ker je rezerviran za oddajanje vsem v omrežju (angl. broadcast).

- *Funkcija* - Vsaka funkcija ima svojo edinstveno kodo, ki je lahko standardizirana ali uporabniško opredeljena. V nadaljevanju se bom osredotočil le na standardni funkciji štiri in šest, ki ju potrebujem za izvedbo funkcionalnosti.

- *Številka telegrama* je kontrolna številka namenjena gospodarju, s katero preveri ali je dobil odgovor na pravilno poslani telegram. Standardni protokol tega podatka ne predvideva!

- *Začetni naslov registra* služi kot referenca na register.

- *Velikost podatkovnega dela* - To informacijo vključi v telegram suženj, ki pove koliko bajtov je v delu telegrama namenjenih podatkom.

Vsebina in velikost podatkovnega dela je odvisna od funkcije. Velikost je navzgor omejena in lahko vsebuje največ 249 bajtov. Zadnja dva bajta predstavljata CRC kontrolno vsoto za detekcijo napak, ki se izračuna na podlagi serije preostalih bajtov v sporočilu.

2.2 Funkcije

Razdelek je namenjen predstavitvi dveh standardnih funkcij, ki razpolagata z registri. V standardu obstaja mnogo funkcij, ki so pogosto uporabljene pri komunikaciji z elektronskimi napravami, vendar niso vključene v implementacijo, ker omenjeni funkciji zadostujeta našim zahtevam.

2.2.1 Funkcija 4

Vprašanje

S to funkcijo odjemalec zahteva od strežnika določene registre, pri čemer ima na voljo 16-bitni naslovni prostor. Odjemalec mora poleg naslova posredovati še podatek o številu zahtevanih registrov, ki se vpiše na podatkovno mesto telegrama. Primer vprašanja gospodarja o registrih na naslovu 39000, kjer zahtevamo dva registra od sužnja 1, prikazuje tabela 2.2.

<table>
<tr><td>Naslov sužnja</td><td>01</td></tr>
<tr><td>Funkcija</td><td>04</td></tr>
<tr><td>Zaporedna številka telegrama</td><td>02</td></tr>
<tr><td rowspan="2">Začetni naslov registra</td><td>98</td></tr>
<tr><td>58</td></tr>
<tr><td rowspan="2">Podatek</td><td>00</td></tr>
<tr><td>02</td></tr>
<tr><td rowspan="2">CRC</td><td>59</td></tr>
<tr><td>5d</td></tr>
</table>

Tabela 2.2: Vprašanje gospodarja o registrih

Odgovor

Suženj odgovori s telegramom, ki vsebuje poleg svojega naslova še število vrnjenih registrov ter njihove vrednosti. Za vsak podatkovni register je predvidenih 16 bitov. V tabeli 2.3 je prikazan odgovor na vprašanje gospodarja.

Naslov sužnja	01
Funkcija	04
Zaporedna številka telegrama	02
Število bajtov	04
Podatki	98 58
	11 18
CRC	09 a1

Tabela 2.3: Odgovor, ki vrne vse zahtevane registre

2.2.2 Funkcija 6

Vprašanje

V prejšnjem razdelku sem z vprašanji zahteval od sužnja, da vrne vrednost določenih registrov. Pri tem potrebujem tudi obratno funkcijo, ki je seveda funkcija pisanja v register.

Oblika telegrama je v grobem analogna telegramu funkcije branja, ki se razlikuje le v podatkovnem delu. Podatek predstavlja vrednost registra, ki ga suženj vpiše v svoje registre.

Odgovor

V primeru uspešnega vpisa v register, suženj oblikuje enak odgovor prejetemu, sicer vrne ustrezno sporočilo o napaki.

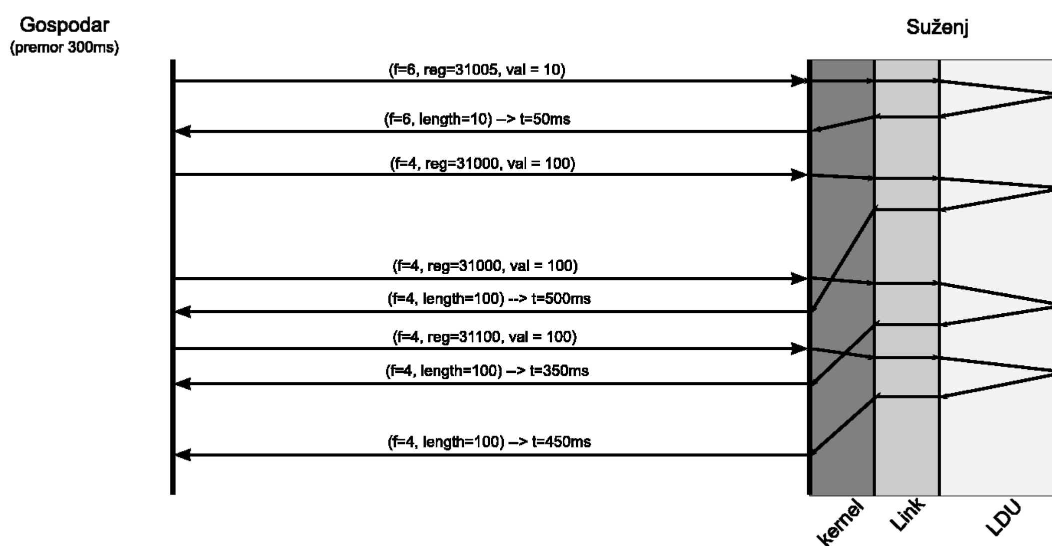
2.3 Modifikacija protokola

V zgodnji fazi razvoja so prvi testi pokazali, da prihaja do nenavadnih napak pri oddajanju telegramov na strani gospodarja. Napaka se je odražala v napačnih odgovorih glede na vprašanje, zato sem v protokol dodal še zaporedno številko telegrama. Pri določenih zahtevah, kjer je število zahtevanih registrov s strani gospodarja preveliko, jih enostavno ni mogoče prenesti vseh naenkrat. V tem primeru gospodar zahteve razdeli in jih sekvenčno pošlje sužnju. Če medtem pride do prevelike zakasnitve odgovora, gospodar ponovi

Naslov naprave sužnja	01
Funkcija	04
Zaporedna številka telegrama	02
Število bajtov	04
Podatki	98 58
CRC	09 a1

Tabela 2.4: Vprašanje in odgovor pri pisanju v register

isto vprašanje, medtem pa dobi odgovor na prejšnje vprašanje in prišlo je do izgube pravilnega vrstnega reda odgovorov, kot je prikazano na sliki 2.1.



Slika 2.1: Zamik vrstnega reda telegramov zaradi zakasnitve.

Če analiziramo vsak telegram posebej, ugotovimo naslednje:

- Poslani telegram vpiše vrednost 10 v register 31005, na katerega dobi takojšen odgovor.
- Telegram zahteva registre 31000-31100, vendar odgovora v predvidenem času 300ms gospodar ne dobi, zato znova pošlje isto zahtevo.

- Telegram prispe, sicer s pravilnim odgovorom, ampak na prejšnjo zahtevo, za katero gospodar misli, da se je izgubila. Do sedaj ni bilo mogoče ugotoviti, da je karkoli narobe z odgovorom, zato gospodar mirno nadaljuje.
- V naslenjem sporočilu zahteva registre 31100-31200. Prispeli odgovor je napačen, saj je v bistvu odgovor na prejšnje vprašanje in ga v protokolu ni mogoče odkriti.

Napake so se pojavljale predvsem pri dolgih telegramih in natančni razlogi niso bili nikoli povsem razjasnjeni. Nedvomno pa je zakasnitev nastajala na nivoju Kernel-a oziroma v samem gonilniku vmesnika RS-232. Naredil sem mnogo testov na preostalih plasteh in izračunani časi obdelave zahteve na posamezni plasti niso nikoli presegli 20ms. Povprečen čas izvajanja celotne zahteve pa je bil okrog 50ms, tako da je bila izločena vsakršna možnost napake na moji programski opremi.

2.4 Arhitektura

Preden preidem na izvedbo protokola, bom za nadaljnje lažje razumevanje predstavil arhitekturo informacijsko komunikacijskega sistema (IKS) imenovanega rcman, pri čemer se bom skušal osredotočiti le na bistvene komponente, relevantne za razumevanje izvedbe protokola Modbus.

V splošni rabi je veliko različnih arhitektur IKS (ISO/OSI, TCP/IP, X.25), ki so v svojem bistvu množica komunikacijskih protokolov. Vsi se delijo na posamezne plasti, na katerih delujejo različni protokoli, ki so zadolženi za opravljanje določenih funkcij.

Število plasti se razlikuje od arhitekture do arhitekture. Arhitektura IKS rcman sestavljajo štiri plasti, kakor jo prikazuje tabela 2.5.

1. **Adapter** - omogoča neposreden dostop do komunikacijskega vmesnika, ki je v našem primeru RS-232.
2. **Link** - skrbi za vzpostavitev sej, oblikovanje in ponovno oddajo sporočil, validacijo, ...
3. **Driver** - izvaja aplikacijske funkcije protokola. Na tej plasti je skupaj s protokolom Modbus realiziran tudi servisni program. Tukaj velja omeniti še gonilnik EventDriver, iz katerega je izveden Modbus in ga razlagam v poglavju 3.8.

Ime plasti	Nivo protokola ISO/OSI	Modbus
DRIVER	5,6	protokol Modbus, servisni program LDU, gonilnik EventDriver
ROUTER	4	-
LINK	2,3	posredovanje telegramov ustrezni aplikaciji
ADAPTER	1,2	branje/pisanje na vmesnik RS-232

Tabela 2.5: Arhitektura rcman

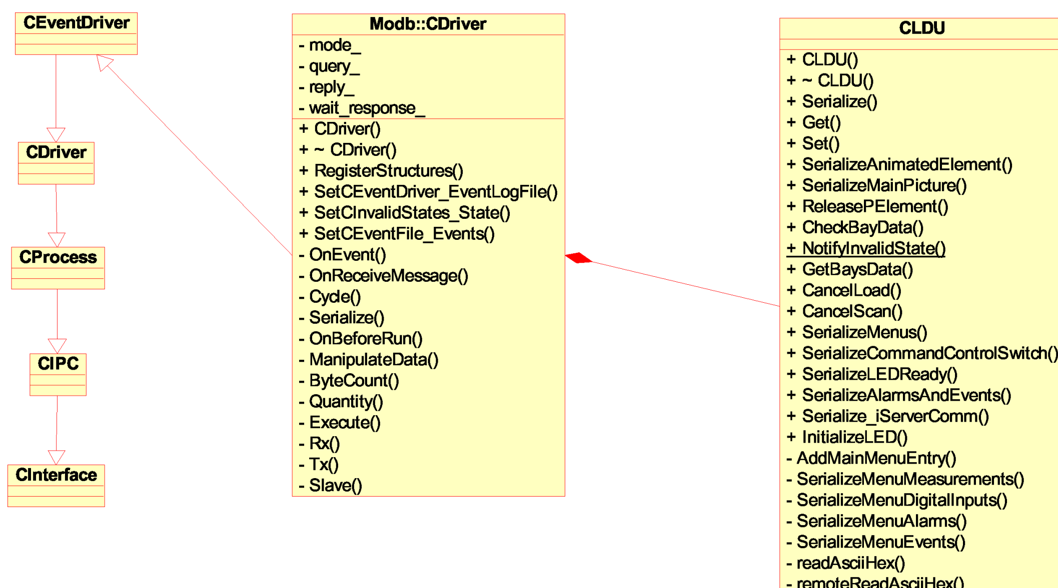
Vse naštete plasti tvorijo celovito funkcionalnost, ki se izvajajo vsaka v svoji procesni niti. Komunikacija med nitmi poteka s pomočjo cevi (angl. pipes), ki jih upravlja rcman. Za poenostavljeno upravljanje in kontrolo stanja procesnih niti, se paralelno izvaja še ukazna nit. Namen slednje nima bistvenega pomena v diplomski nalogi, zato samo omenim, da se prek nje izvaja nadzor stanja nad procesnimi nitmi, kot je: zaganjanje, ustavljanje, prekinjanje in podobno.

2.5 Izvedba protokola

Modul protokola Modbus je izveden po vzoru ostalih gonilnikov sistema rcman. Fizično je vsak modul v obliki dinamične knjižnice, ki se kot vtič (angl. Plug-in) vključi ob inicializaciji rcman-a.

Statični model protokola prikazuje razredni diagram 2.2, s katerega je razvidna hierarhija razredov. Vsi razredi v hierarhiji nižje od `CEventDriver` so abstraktni razredi, ki narekujejo vmesnike za implementacijo gonilnikov.
Na diagramu nista prikazani dve zunanji (angl. extern) funkciji, kjer se modul instancira in služita kot vhodni točki v modul. Namen obeh funkcij je inicializacija procesne in ukazne niti, ki jih kliče rcman. S stališča modula je pomembna le procesna nit, saj se v njej izvaja njegova funkcionalnost.

Modbus je neposredno izveden iz razreda `CEventDriver`, pri katerem sem moral redefinirati pomembnejše funkcije vmesnika `Plug::CInterface`, kot so `OnReceiveMessage`, `OnEvent`, `Cycle` in `Serialize`, ki jih po določenem zaporedju kliče rcman. Kot bom opisal v nadaljevanju, je za Modbus najvažnejša funkcija `OnReceiveMessage`, njen namen pa je sprejemanje sporočil iz nižje ležeče plasti.



Slika 2.2: Hierarhija razredov za izvedbo protokola Modbus

2.5.1 Sprejem, oddaja in dekodiranje telegramov

Iz kode 1 je razvidno, da v funkciji `Interface` modul Modbus instanciramo in vrnemo rcman-u. S tem sem zagotovil referenco na modul z ustreznim vmesnikom. Vhodna točka sprejema sporočila iz nižje ležeče plasti je virtualna funkcija `OnReceiveMessage`, ki v argumentu vsebuje podatke o sporočilu in je v mojem primeru telegram. Sporočilo je v predpisanem formatu, zato ga je potrebno najprej dekodirati in preveriti redundanco. Vsebina telegrama se shrani v strukturo `ADU`, ki vsebuje vse tiste pripadnike, ki jih narekuje standard Modbus. Več o samem pomenu pripadnikov je razloženo v razdelku 2.1.

Koda 1 - Struktura vsebuje podatke telegrama

```
struct ADU{
  unsigned char station;
  unsigned char function;
  unsigned char id;
  unsigned char data[252 + 2];
  short length;
};
```

Koda 2 - Struktura vsebuje podatke telegrama

```
// Create command thread
extern "C" bool Start ( Plug::Handle a_server ){
}
// Create process thread
extern "C" Plug::CInterface* Interface ( Plug::Handle a_server,
                                         Plug::Handle a_client,
                                         fd_set* a_pfd_set){
  LDUModb::CDriver* driver = new LDUModb::CDriver ( a_server );
  return driver->Interface ( a_client, a_pfd_set );
}
```

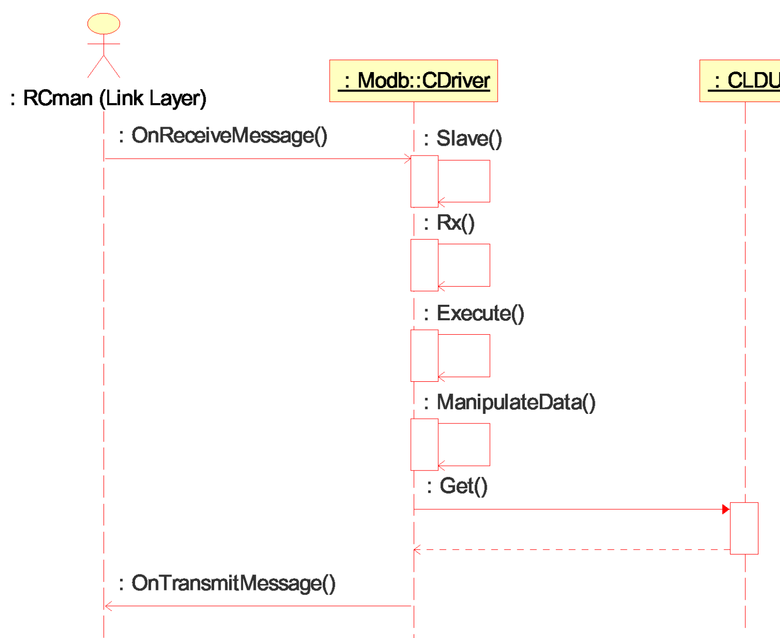
Ko je telegram dekodiran, se začne izvrševanje zahtevane funkcije. Naj spomnim, da protokol podpira le funkciji pisanja in branja, ki se na osnovi zahtevanega registra v `CLDU` instanci poišče ustrezen objekt in pokliče funkcija `Get()` ali `Set()`. Več o omenjenih objektih je predstavljeno v poglavju 3. Po končani izvršitvi funkcije se oblikuje še ustreznem odgovor, ki je odvisen od zahtevane funkcije. Nazornejšo dinamiko funkcijskih klicev prikazuje sekvenčni diagram 2.3.

2.6 Testiranje

Standardni del razvojnega procesa vključuje testiranje funkcionalnosti. Pogosto se ti testi imenujejo enotski (angl. unit) testi, kjer razvijalec postopno testira svoje programske enote oziroma funkcionalnosti, preden jih preda testni ekipi. Slednja vse funkcionalnosti znova pretestira in na koncu naredi integracijski test. V primeru hroščev se posreduje ustrezna povratna informacija razvijalcu, ki po odpravi napake ponovi celoten postopek.

Za hitro in učinkovito enotsko testiranje je potrebno zagotoviti ustrezno okolje, kjer je možno testiranje novih funkcionalnosti in odpravljanje hroščev brez nepotrebnih dodatnih opravil. To še zlasti velja za projekte, kjer je ciljna platforma različna razvojni, pri kateri se veliko časa porabi za zamudno prenašanje novih datotek. Velikokrat se naprava znajde v nekonsistentnem stanju in posledično je potreben ponovni zagon naprave, kar terja veliko dragocenega časa.

Večina testiranj je možno na razvojni platformi, saj se funkcionalnosti v 99% obnašajo enako kot na ciljni. To seveda ne velja za časovno kritične (angl.

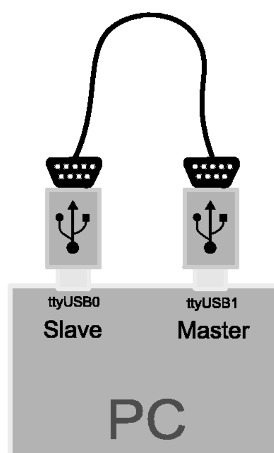


Slika 2.3: Sekvenčni diagram dekodiranja sporočila

Real-time) aplikacije, za katere sicer obstajajo razni simulatorji okolja, vendar niso kos časovnim zahtevam.

Testiranje servisnega programa sem izvajal na razvojni platformi, kjer so prvotni problem predstavljali serijski vmesniki (RS-232). Ker današnji računalniki niso več opremljeni s tovrstno strojno opremo, sem moral izbrati funkcijsko nadomestljiv in danes najpopularnejši vmesnik USB. S preprostim trikom dveh USB-RS232 pretvornikov in serijskega kabla, sem prišel do željenega rezultata, ki povezuje gospodarja in sužnja med seboj, kot ga prikazuje slika 2.4. Rešitev se je izkazala za zadovoljivo, kljub temu, da je v redkih primerih prihajalo do popačenih ali celo izgubljenih telegramov.

Za še hitrejše testiranje servisnega programa sem si ustvaril pomožno aplikacijo, ki simulira obnašanje gospodarja. S tem sem si zagotovil neodvisnost od prikazovalne enote, ki je predstavljala vir potencialnih problemov in časovnih obremenitev. Tako sem s pomožno aplikacijo generiral poljubne telegrame v različnih razmerah. Testiranje s prikazovalno enoto se je izvajalo šele v zadnji fazi posamezne razvojne iteracije in je služil kot končni enotski test funkcionalnosti.



Slika 2.4: RS-232 zanka s pomoč dveh USB pretvornikov

2.7 Pomankljivosti in izboljšave

Izvedba je šla skozi tri razvojne iteracije in v začetni fazi mnogo funkcionalnosti ni bilo definiranih, niti jih ni bilo mogoče predvidevati. Pri tem je najbolj trpela zasnova nastale programske opreme, ki ni ravno vzorna in jo ne morem predstavljati v superlativih. Zagotovo se bo morala popraviti v prihodnji razvojni iteraciji, saj je obstoječa zelo težko nadgradljiva. Zaradi slabe zasnove je vmesnik protokola praktično neuporaben in ne dopušča ponovne uporabe v drugih aplikacijah, ne da bi pri tem spreminjali implementacijo. Po drugi strani pa specifikacija ni predvidevala ponovne uporabe kode, zato je izvedba vendarle sprejemljiva.

Druga, ne tako očitna neustreznost zasnove, je lokacija registrske baze. Logično je za pričakovati, da bi se registrska baza nahajala v razredu Modbus, vendar zaradi načina izvedenosti funkcionalnosti LDU in lažje manipulacije registrov, se dejansko nahaja v `CLDU` razredu.

Poglavje 3

Funkcionalnosti lokalne prikazovalne enote

Osnovni namen servisnega programa je zadostitev zahtev s strani gospodarja. Večinoma so to zahteve po informacijah, ki odražajo dejansko stanje sistema v določenem trenutku. Število realiziranih funkcionalnosti je omejeno, zato se jih mora gospodar zavedati in poznati način, kako do njih dostopati.

V servisni program sem implementiral tiste funkcionalnosti, ki jih je predvidevala specifikacija v posamezni razvojni iteraciji. Pred izvedbo sem moral preučiti določene module in vmesnike v sistemu, ki služijo kot viri podatkov. Šele zatem sem lahko implementiral funkcionalnosti, kot so: analogne meritve, izvrševanje ukazov, stanje digitalni vhodov, alarme, dogodke ter indikatorji LED. Skozi poglavje bom skušal nazorno predstaviti pomen omenjenih funkcionalnosti, njihovo izvedbo ter prilagajanje.

Za pravilno delovanje dogodkov in alarmov skrbi gonilnik EventDriver, zato izvedba vključuje tudi njegovo integracijo. Zaradi neustrezne obstoječe rešitve sprejema dogodkov v omenjenem gonilniku, sem bil primoran opraviti ustrezno predelavo in zagotoviti stabilnost, ne samo servisnemu programu in EventDriver-ju, temveč tudi ostalim modulom v sistemu.

3.1 Načrtovanje in izvedba zasnove

V prvi razvojni iteraciji je bilo število funkcionalnosti malo, saj so bile vključene le analogne meritve in shema polja, zato sem se raje odločil implementirati s kompozicijo in ne z dedovanjem. V zadnji razvojni iteraciji se je to izkazalo kot slabost, saj so prišle dodatne funkcionalnosti, ki so zahtevale integracijo gonilnika EventDriver. S tem sem si povzročili probleme s prenašanjem podat-

kov med obema funkcionalnostima, kar je na koncu privedlo do treh osnovnih problemov načrtovanja in izvedbe:

- manipulacija registrske baze (branje/pisanje),
- zagotavljanje enovitosti instanc,
- integracija gonilnika EventDriver.

Slednjega obravnavam v sklopu funkcionalnosti dogodkov in alarmov, ker na takšen način lažje razložim problem in njegovo rešitev.

3.1.1 Manipulacija registrske baze

Nedvomno najtežji problem pri načrtovanju je predstavljalo učinkovito serviranje zahtev branja in pisanja registrske baze. Pri tem sem izhajal iz stališča, da omogočim čim enostavnejše dodajanje novih funkcionalnosti. Ta namen je bil dosežen z uporabo objektnega pristopa in s tem izkoristil prednost programskega jezika C++. Odjemalec se tako ne zaveda konkretnega tipa objekta s katerim razpolaga, temveč samo njegovega vmesnika, ki ga mora konkretni razred implementirati.

Vsaka funkcionalnost obsega n število registrov, ki so določeni na intervalu $[p, k]$, pri čemer je $\{k = p + n, p_{min} = 0 \wedge k_{max} = 2^{16} - 1\}$. Problem sem rešil tako, da uporabim parametriziran razred `std::map` s ključem `CLDUAddress`, ki hrani objekte z vmesnikom `CLDUElement`, kateremu sem priredil funktor primerjave `LessThan`. Iz abstraktnega razreda `CLDUElement` so izvedeni vsi objekti, ki pokrivajo določeno funkcionalnost. Elementi se nahajajo v objektu `map_`, do katerih je mogoče dostopati z ustreznim ključem tipa `CLDUAddress`. Ta je določen z intervalom registrov, ki ga servisni program dobi iz protokola Modbus in na katerem sloni funktor primerjave. Enostaven primer opisane implementacije je predstavljen na sekvenčnem diagramu v dodatku A.1.

Interval registrov je vnaprej dogovorjena specifika z implementatorjem programa na prikazovalni enoti (angl. firmware), ki so v nadaljevanju jasno opredeljeni za vsako funkcionalnost posebej.

3.1.2 Vzorec singleton in večpoljska podpora

V OOP (Object Oriented Programming) se včasih znajdemo v situaciji, v kateri želimo zagotoviti enovitost enega ali več objektov. Za dosego takšnega cilja obstaja vzorec singleton, ki ima eleganten način implementacije vmesnika. Instanciranje objekta se izvede posredno prek statične funkcije `Instance`, ki

Koda 3 - Razredi, ki skrbijo za dostop do registrske baze

```
/** Abstraktni razred Element  */
class CLDUElement{
  public:
    CLDUAddress address_;
    /* Kazalec na bazo registrov */
    unsigned char* pparent_;
    CLDUElement (unsigned char* a_pparent) : pparent_(a_pparent);
    /* Funkcija branja registrov */
    virtual bool Get() { return true; }
    /* Funkcija pisanja v register */
    virtual bool Set() { return true; }
};
/** Primerjalni funktor */
class LessThan : public std::binary_function<const CLDUAddress&,
                                             const CLDUAddress&,
                                             bool>{
  public:
  bool operator () (const CLDUAddress& a_address,
                    const CLDUAddress& b_address) const{
    return ((a_address.begin_+a_address.size_-1)<b_address.begin_);
  }
};
/* Deklaracija kontejnerja */
std::map<CLDUAddress, CLDUElement*, LessThan> map_;
```

Koda 4 - Implementacija singleton vmesnika

```
class CLDUSingleton{
  public:
    static CLDUSingleton* Instance(unsigned char* pregs);
    static void Destroy();
  protected:
    CLDUSingleton( unsigned char* pregs );
    static CLDUSingleton* pinstance;
}
```

instancira objekt v primeru neobstoja, sicer vrne obstoječega. Da sem preprečil uporabniku neposredno instanciranje objekta, sem konstruktor ustrezno zaščitil.

Nekaj pozornosti velja še nameniti sproščanju resursov. Ni pravilno razmišljanje, da sprostitev objekta izvedemo s standardnim destruktorjem, saj nas lahko v določenih primerih privede do težav. Če več razredov implementira isti singleton in se izvede sprostitev različnih objektov, pride do večkratne sprostitve istega dela pomnilnika. To lahko privede do nestabilnosti in v najpogostejšem primeru do razsutja aplikacije. Naredil sem test obeh različic in prišel do ugotovitve, da aplikacija ne preživi na ciljnem sistemu brez statičnega destruktorja. Testiral sem tudi na razvojni platformi, na kateri aplikacija čudežno preživi.

V gonilniku je singleton implementiran za realizacijo večpoljske (multibay) podpore. Specifikacija iz prve iteracije je zahtevala izvedbo prikaza stanj več različnih naprav na enem prikazovalniku. Iz tega izhaja beseda večpoljska, kjer je vsako polje lokalna ali oddaljena naprava. Aktivno polje predstavlja instanco singleton, s čimer sem zagotovil poenostavljeno preklapljanje med posameznimi polji. Zaradi počasnega odziva oddaljenih polj, so se v zadnji iteraciji začasno onemogočila, kljub temu pa implementacija funkcionalnosti to še vedno omogoča.

3.2 Viri podatkov

Pomembna lastnost dobre programske opreme je njena fleksibilnost, saj le tako omogočimo uporabniku prilagajati obnašanje programske opreme brez predhodnega prevajanja. Vhodni podatki v svoji pojavni obliki morajo obstajati na nekakšnem fizičnem mediju (trdi disk, flash, dvd,...). Obstaja na ducate oblik hranjenja podatkov, iz katerih je potrebno izbrati tistega, ki najbolj ustreza postavljenim kriterijem. Vsi nimajo enake teže, zato mnogokrat vplivajo na odločitev razvojni viri, ki nimajo neposredne povezave z uporabnostjo in učinkovitostjo. Kljub dejstvu, da relacijska baza predstavlja najučinkovitejši način hranjenja in dostopa do podatkov, je po drugi strani implementacija sistema za upravljanje s podatkovno bazo (SUPB) vse drugo prej kot enostavna. Odlično alternativo predstavlja baza XML, saj je za razliko od SUPB-ja implementacija analizatorja relativno enostavna.

3.2.1 XML

Označevalni jezik XML omogoča strukturirano opisovanje lastnosti na podoben način kot razredi v katerem od objektno usmerjenih programskih jezikov. Strukturirani podatki so tako na voljo aplikacijam, ki služijo kot podatkovna baza oziroma vir podatkov.

Prednost XML notacije se izkaže v že omenjeni relativno preprosti implementaciji analizatorja. Glede na namembnost in način procesiranja dokumenta, sta najvidnejša analizatorja DOM in SaX. Takšno procesiranje podatkov se imenuje serializacija podatkov. Serializacija je postopek pretvarjanja objekta v zaporedje bitov, ki jih je možno shraniti v pomnilnik, trdi disk ali pa jih pošljemo po omrežju. Inverzni proces od serializacije se imenuje deserializacija, pri katerem se objekt iz serije bitov sestavi v prvotno obliko. Več o serializaciji/deserializaciji servisnega programa je predstavljeno v razdelku 3.3.

V sistemu rcman deluje analizator DOM, ki celotno strukturo dokumenta analizira in ustvari ustrezno drevesno strukturo. Slabost takšnega načina se pokaže pri velikih dokumentih, saj se celotna struktura nahaja v fizičnem pomnilniku. Takšna slabost se še posebej izraža na napravah, kjer je količina fizičnega pomnilnika majhna.

Programski moduli so striktno odvisni od strukture XML dokumenta, ki v primeru nepravilnosti vodi do nepravilnega delovanja modula. Struktura ni samoumnevna, zato je potreben način za zagotavljanje njene pravilnosti. Strukturo dokumenta XML narekuje pripadajoča shema, kjer so opredeljeni vsi možni gradniki, njihovi tipi ter omejitve. Na njihovi osnovi se v namenskih urejevalnikih zagotovi pravilnost dokumenta XML in ustrezno strukturirane vhodne podatke aplikacijam.

3.2.2 Atomična baza

Dinamični vir podatkov, ki jih uporabljajo funkcionalnosti LDU, je atomična baza. Kot samo ime pove, so osnovni gradniki atomi, ki vsebujejo reference na registre in njihove vrednosti. Za posodabljanje vrednosti registrov skrbi real-time modul, ki slika dejanska stanja registrov. V podrobnosti atomične baze in njene uporabe se ne bom poglabljal, zato na tej točki le omenim, da funkcionalnosti v servisnem programu uporabljajo atomično bazo zgolj za namene branja registrov. V atomično bazo oziroma register, servisni program nikoli ne piše, razen prek posrednikov npr. v ukazih.

3.3 Serializacija

Statični vhodni parametri, ki so relevantni pri serializaciji modula, se nahajajo v bazi XML. V tem sklopu se v bazo objektov, ki so zadoloženi za osveževanje registrov, ustvarijo ustrezne instance, zato ne sme priti do napak. V splošnem velja pravilo, da se nastavijo vsi parametri za posamezno funkcionalnost, ker v nasprotnem primeru lahko ogrozimo stabilnost modula ali pa se privede do nepravilnega delovanja. Serializacija se izvede zaradi večpoljske podpore v dveh stopnjah:

- inicialna
- poljska (angl. bay)

Inicialna serializacija se izvede ob zagonu servisnega programa, ki instancira funkcionalnost LDU in izvede ustrezen funkcijski klic. Vključene so vse tiste komponente, ki so neodvisne od posameznih polj, kot so: polja v lokalnem omrežju, animirani elementi ter delno alarmi in dogodki.

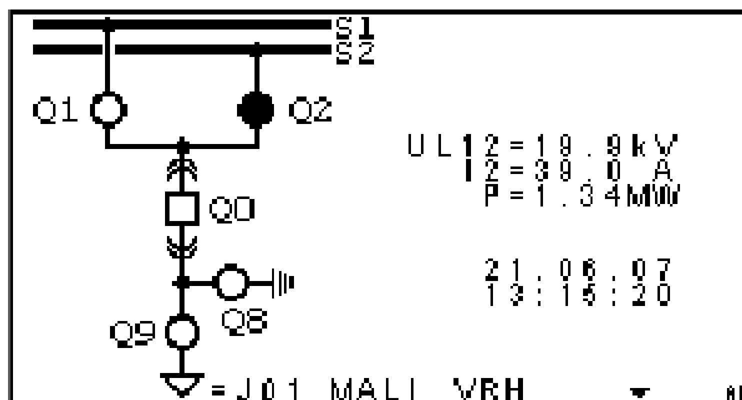
Druga faza se izvede šele s prvim telegramom, ko gospodar zahteva prvo (lokalno) polje, pri čemer se serializirajo vsi ostali podatki.

3.4 Osnovna stran

Po uspešni komunikaciji gospodarja s sužnjem in serializaciji, se za izbrano polje naloži osnovna stran, v katerem se prikazujejo:

- slepa (neaktivna) shema polja,
- animirani elementi,
- ime polja,
- osnovne meritve,
- ura in datum.

V nadaljevanju bom opisal pomen in izvedbo naštetih funkcionalnosti ter pomen registrov v protokolu.



Slika 3.1: Osnovni meni na prikazovalniku

3.4.1 Slepa (neaktivna) shema polja

Shema polja prikazuje, na kakšen način so posamezni izvršilni elementi med seboj povezani. To omogoča zaščitnemu inženirju jasen pregled in lažje upravljanje z elementi.

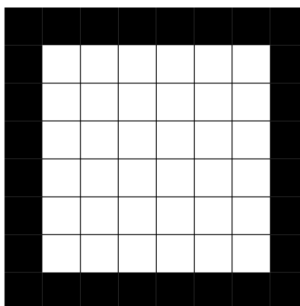
Slika sheme je v binarni obliki, ki je določena z višino h in širino w, ki jo zapišemo kot $w \times h$. Za posamezno točko velja:

$$p(x, y) = \begin{cases} 1 & \text{viden objekt} \\ 0 & \text{prazna površina} \end{cases}$$

Binarna slika v datoteki vsebuje šestnajstiške vrednosti, kjer vsaka predstavlja osem bitno število. Prednost tega zapisa je v performansah, saj ni potrebno nikakršnih pretvorb in v prenosu manjšega števila podatkov prek serijskega vmesnika, saj vemo, da je telegram sestavljen iz enot v velikosti osmih bitov. Primer zapisa izvršilnega elementa v datoteki, ki predstavlja binarno sliko:

```
0xFF,0x81,0x81,0x81,0x81,0x81,0x81,0xFF
```

Bistven podatek za pravilen prikaz slike je širina in višina slike, ki ju dobim iz serializacije. S tem lahko določim dimenzijo slike in uspešno interpretiram posamezne bajte. Zgornji primer 8×8 je trivialen, saj vsaka vrednost predstavlja po eno vrstico. Kljub temu bi brez podatka o višini in širini lahko predpostavljal, da gre za sliko velikosti $16 \times 4{,}32 \times 2{,}64 \times 1$ in obratno. Pretvorbo zgornjega zapisa v ničle in enice prikazuje slika 3.2.



Slika 3.2: Binarna slika

Izvedba

Osnovna slika je statična in zanjo ni potrebno nikakršno osveževanje, zato se slika v registrsko bazo vpiše v fazi serializacije polja. Slika se za lokalno polje nahaja v datoteki na statičnem pomnilniku, iz katerega se v funkciji `readAsciiHex` preberejo vse šestnajstiške vrednosti in vpišejo v registrsko bazo.

Registri protokola

V protokol je podprta funkcija branja.

Register	Opis (Hi)	Opis (Lo)
55000(0xD6D8)	X_hi(položaj)	X_lo(položaj)
55001	Y_hi(položaj)	Y_lo(položaj)
55002	H_hi(velikost)	H_lo(velikost)
55003	W_hi(velikost)	W_lo(velikost)
55004	rezervirano	število elementov
55005	⋮	⋮
55009	rezervirano	rezervirano
55010(0xD6E2)	podatek	podatek
55005	⋮	⋮
65499	0	0

- **X,Y položaj** - X,Y koordinata začetne točke osnovne slike. Koordinate so izražene v točkah zaokrožene na 8 bitov.
- **H,W velikost** - Širina in višina osnovne slike. Velikost je izražena v točkah zaokrožene na 8 bitov.

- **število elementov** - Podatek o številu animiranih elementov na sliki, ki ga potrebuje gospodar.

- **podatek** - Tukaj se nahajajo podatki o sliki v bajtnem načinu po posameznih vrsticah, od leve proti desni strani prikazovalnika. Velikost slike je odvisna od velikost prikazovalnika.

3.4.2 Animirani elementi

V prejšnjem razdelku sem predstavil statično sliko, ki povezuje posamezne izvršilne elemente med seboj. Na mestih, kjer je predviden prostor za elemente, so običajno prazni prostori, tako da se izognemo popačenju slike zaradi možnih vidnih delov sheme polja.

Prikaz okvirja (angl. frame) je odvisna od vrednosti registra v atomični bazi, ki odraža dejansko stanje izvršilnega elementa. Večina elementov vsebuje po en bit za indikacijo statusa, zato imajo takšni elementi najmanj dva okvirja.

Slika 3.3: Primer animiranih okvirov ločilke v različnih stanjih

Izvedba

Animacije zasedejo relativno veliko prostora, zato se prenesejo na prikazovalno enoto samo enkrat. V registrsko bazo se vpišejo v inicialni serializaciji in se ne osvežujejo, podobno kot pri shemi polja, zato ni potrebne nobene interakcije. Statusi animiranih elementov referirajo na animacijo s pomočjo indeksa, ki je odvisen od zaporedja serializacije.

Registri protokola

Za animirane elemente je v protokolu podprta funkcija branja, pri čemer so registri deljeni na

- registre animiranih elementov in

- registre statusov.

V registre lahko spravimo največ 6 elementov, ki predstavljajo množico razpoložljivih animiranih elementov. Vsak element pa ima lahko največ štiri okvirje. Pri tem je vsak okvir omejen na 32 bajtov, kar omogoča maksimalno velikost elementa 16×16. Poleg podatkov o animaciji, se v registrih nahaja tudi velikost posameznega okvirja.

Register	**Opis (Hi)**	**Opis (Lo)**
Animirani element 0, okvir 0		
45000(0xAFC8)	H_hi(velikost)	H_lo(velikost)
45001	W_hi(velikost)	W_lo(velikost)
45002	rezervirano	rezervirano
45003	⋮	⋮
45005	podatek	podatek
45003	⋮	⋮
45199	0	0
Animirani element 0, okvir 1		
45200	⋮	⋮
45399	0	0
Animirani element 5, okvir 3		
49600	⋮	⋮
49799	0	0

- **H,W velikost** - Širina in višina animiranega elementa oziroma okvirja. Velikost je izražena v točkah zaokrožene na 8 bitov.

- **podatek** - V teh registrih se nahajajo podatki o sliki za posamezni okvir.

3.4.3 Statusi animiranih elementov

Registri statusov so dinamični in iz tega razloga ustrezno ločeni od animacij. S tem sem olajšal osveževanje registrov in prenos nepotrebnih podatkov po serijskem vmesniku. Če bi bili statusi vrinjeni med registre animacij, bi morali poleg statusov elementov prenašati še podatke o animacijah, ker ni možno drugače, kot prenos celotnega intervala registrov naenkrat.

Izvedba

Animirani elementi se serializirajo v poljski serializaciji, kjer se za vsak element ustvari instanca `CLDUAnimatedElementReference` in se vpiše v seznam statične dolžine. Ker gre za dinamično funkcionalnost, se vpiše tudi v globalno mapo objektov. Ob zahtevi gospodarja se poišče ustrezni objekt elementa in pokliče pripadajočo funkcijo `Get`, ki neposredno iz atomične baze prebere vrednosti registrov.

Registri protokola

Za ažurne podatke se statusi kličejo ciklično s strani gospodarja. Poleg statusov se v registrih nahajajo še položaji animiranih elementov. Dejansko največje število elementov, ki jih lahko prikazuje prikazovalnik je 50, in jih je možno parametrirati v raznih kombinacijah iz množice animiranih elementov.

Register	Opis (Hi)	Opis (Lo)
Statusi animiranega elementa 0		
44000(0xABE0)	X_hi(položaj)	X_lo(položaj)
44001	Y_hi(položaj)	Y_lo(položaj)
44002	rezervirano	čas za ukaz
44003	status elementa	element
44004(0xABE4)	status ukaza	izdaja ukaza
Statusi animiranega elementa 1		
44005(0xABE5)	X_hi(položaj)	X_lo(položaj)
44006	⋮	⋮
44009(0xABE9)	status ukaza	izdaja ukaza
Statusi animiranega elementa 24		
44120(0xAC58)	X_hi(položaj)	X_lo(položaj)
44121	⋮	⋮
44124(0xABE4)	status ukaza	izdaja ukaza

Tabela 3.1: Statusi animiranih elementov

- **X,Y položaj** - Koordinate začetne točke animiranega elementa. Koordinate so izražene v točkah zaokrožene na 8 bitov.
- **Čas za ukaz** - Potreben čas za izvedbo ukaza elementa primarnega EES.

- **Status elementa** - Stanje animiranega elementa se nastavi glede na vrednost atoma oziroma registra:

Status elementa			
Bit 15	...	**Bit 9**	**Bit 8**
rezervirano	...	**status**	**status**

  - stanje **00** = nedefiniran
  - stanje **01** = zaprto
  - stanje **10** = odprto
  - stanje **11** = napaka na elementu

- **Element** - Indeks animiranega elementa, ki je definiran v bazi oziroma registrih animiranih elementov.

Ostala dva registra, ki vsebujeta podatke o ukazih, sta opisana v naslednjem razdelku.

3.4.4 Ukazi

Osnovna funkcionalnost naprave, ki sem jo že omenili v uvodu, je izdajanje ukazov izvršilnim elementom. Za zaščitnega inženirja je to zelo uporabna funkcionalnost ob popravilih in vzdrževanjih, saj ni potrebna komunikacija s centri vodenja za izdajanje ukazov nad posameznimi izvršilnimi elementi.

Pri izdajanju ukazov se posveča posebna pazljivost, saj lahko z ne prav veliko truda poškodujemo ali celo uničimo izvršilni element. To se lahko zgodi v primeru, če brezglavo pošiljamo ukaze na istega. Takšno početje je kljub strokovnu usposobljenosti inženirja možno, če mu ne zagotovimo ustrezne povratne informacije o rezultatu izvedbe ukaza.

Izvedba

Serializacija se izvede v sklopu animiranih elementov, na katerih se izvajajo ukazi. Funkcionalnost v serializaciji zahteva podatke o ukaznih objektih, ki je nujen parameter pri izdaji ukazov. Na vsak animirani element lahko parametriramo ukazni objekt, vendar ni nujno, saj v določenih primerih ne želimo izvajati ukazov nad izvršilnim elementom. V splošnem obstajata “Single pole”in ”Double pole”ukazna objekta, ki se razlikujeta le v številu bitov. Njuna uporaba pa je odvisna od vrste izvršilnega elementa ali ukaza.

Izdajanje ukazov se izvede na zahtevo funkcije pisanja, kjer se kliče funkcija `Set` nad izbranim animiranim elementom. Pri izdajanju sem odvisen od ostalih modulov v sistemu, pri čemer je na voljo ustrezen vmesnik v sistemu.

Status izdanega ukaza se sporoča gospodarju na zahtevo tako, kot pri statusih animacij. S pomočjo atomične baze sem priskrbel gospodarju povratno informacijo, ki prepreči vsakršno ponovno izdajo, če status ni ustrezen.

Registri protokola

V protokolu sta podprti funkciji pisanja in branja. Prva se uporablja za izdajanje ukazov in druga za pridobivanje statusov. Izdajnje ukazov nad posameznim elementom je možno ob dveh predpogojih:

- **možnost izvedbe ukaza**, saj ni nujno, da je možno izvajati ukaze nad vsakim elementom.
- **lokalni način vodenja**, ki se v nasprotnem primeru preda oddaljenemu oziroma centru vodenja.

Registri se nahajajo v sklopu animiranih elementov, ki so opisani v prejšnjem razdelku. Registra za izdajo ukaza in status povratne informacije se nahajata v registru 44004.

Register za izdajo ukaza					
Bit 7	...	**Bit 3**	**Bit 2**	**Bit 1**	**Bit 0**
rezervirano	rezervirano	rezervirano	izdaja ukaza		

- stanje $0x00$ = ni ukaza s strani master naprave
- stanje $0x01$ = ukaz za vklop, izdana s strani naprave LDU
- stanje $0x02$ = ukaz za izklop, izdana s strani naprave LDU
- stanje $0x05$ = ukaz za niže
- stanje $0x06$ = ukaz za više

Register statusa zadnje izdane ukaze					
Bit 7	...	**Bit 3**	**Bit 2**	**Bit 1**	**Bit 0**
rezervirano	rezervirano	rezervirano	status ukaza		

- stanje $xxxxx000$ = ukaz ni bil izdan
- stanje $xxxxx001$ = ukaz uspešno izdan

- stanje $xxxxx010$ = ukaz ni uspel
- stanje $xxxxx100$ = ukaz blokiran med izvajanjem
- stanje $0xxxxxxx$ = ukaz ni možen
- stanje $1xxxxxxx$ = ukaz možen

3.5 Preklop vodenja

V prejšnjem razdelku sem razložil, da je za izdajanje ukazov prek prikazovalne naprave mogoče, če deluje v lokalnem načinu. Razlogi za preklapljanje med lokalnim in oddaljenim načinom so zaradi morebitnih konfliktnih situacij s centrom vodenja. Tipičen primer je rutinski pregled ali vzdrževanje, kjer je potreben odklop, npr. ločilke prek LDU, za tem pa bi se izdal ukaz za vklop iz centra vodenja in s tem bi lahko povzročili katastrofalne posledice za vzdrževalca.

Izvedba

Implementacija preklopa vodenja je zelo podobna kot pri animiranih elementih, s tem da je ukaz preklopa vodenja vedno enobitni (SBC). V serializaciji se na osnovi parametracije ustvari ukazni objekt, s katerim se kasneje upravlja preklop.

Registri protokola

V protokol je vključena samo funkcija pisanja, torej izdajanje ukazov preklopa vodenja. Ker je preklop tudi ukaz, je le-ta lahko blokirana s strani višjih nivojev in se na LED indicira enako kot pred izvršitvijo preklopa.

Register	Opis (Hi)	Opis (Lo)
503	rezervirano	vodenje

Ker obstaja več načinov vodenja, je lahko vrednost registrov naslednje:

- $0x00$ - Nedefinirano vodenje (napaka) ali vsi ukazi blokirani.
- $0x01$ - Lokalno vodenje z zapahovanjem (latched).

- $0x02$ - Daljinsko vodenje z zapahovanjem.
- $0x04$ - Lokalno vodenje brez zapahovanja (non-latched).

3.6 Analogne meritve

Zajem meritev se izvaja prek merilnih vmesnikov primarnega EES, na katerega je priklopljen zaščitni rele. To so meritve tokov, napetosti, moči in energije, ki predstavljajo uporabne informacije za zaščitnega inženirja.

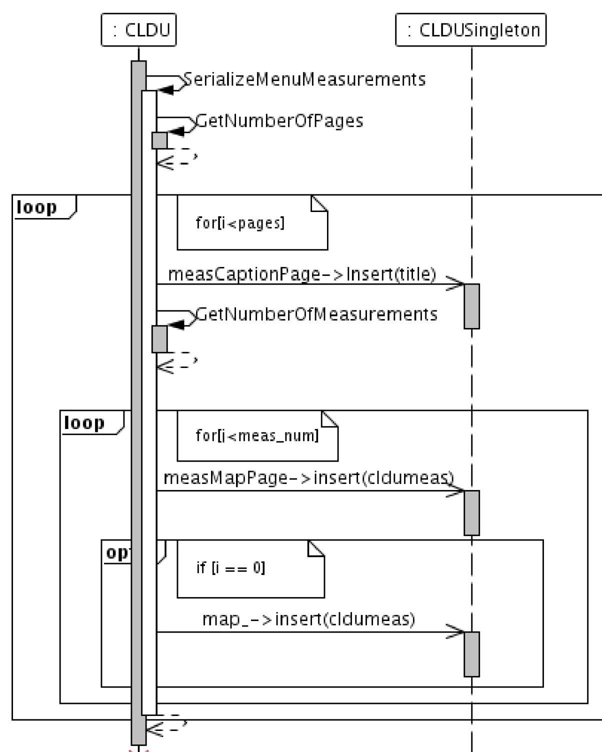
Na prikazovalniku so na voljo tri strani za meritve. Na vsako strani je mogoče prikazati v dveh stolpcih po štirinajst meritev. Posamezna meritev vsebuje parametrirane atribute, kot so:

- ime meritve,
- referenca na atom, ki vsebuje analogno meritev,
- enota,
- število decimalnih mest,
- nominalna in primarna nominalna vrednost,
- faktor.

Izvedba

Podatki o posameznih meritvah se nahajajo v razredu `CLDUMeasurement`, ki se ob serializaciji vpišejo v seznam statične dolžine $3 * 28 = 84$. Poleg tega pa obstaja še ločen seznam dolžine tri za imena menijev. Kljub temu pa se v globalno mapo objektov ne morejo vpisati vse meritve, saj je zaradi omejenega pomnilnika naprave LDU na voljo le za 28 meritev. Zaradi tega sem ustvaril dodaten razred, ki sem ga prav tako vnesel v globalno mapo in služi za menjavanje objektov meritev. V serializaciji so po privzetem v globalno mapo vnešeni objekti, ki pokrivajo prvo stran meritev in se na zahtevo spremembe strani gospodarja ustrezno zamenjajo.

Format meritve za prikaz na napravi LDU je lahko poljuben, saj mu ga posredujem kot sekvenco znakov ASCII. Analogna meritev in nominalni vrednosti se prebereta iz atoma, spremljajoča enota pa je statična. Preden oblikujemo



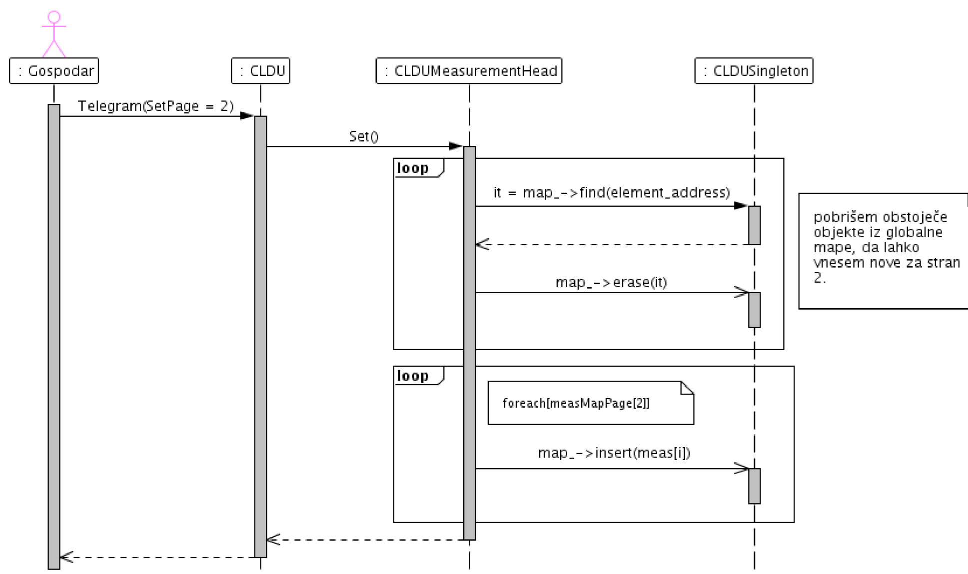
Slika 3.4: Serializacija analognih meritev

meritev v niz znakov, je potrebna pretvorba glede na nominalno vrednost po naslednji formuli:

$$C = primary_nominal * \frac{real_value}{nominal} * factor$$

Registri protokola

V protokol sta vključeni tako funkcija branja kot pisanja. Gospodar piše v register v primeru, kadar želi spremeniti stran na kateri se nahaja. Funkcijo branja pa gospodar kliče ciklično, saj se vrednosti meritev dinamično spreminjajo.



Slika 3.5: Sprememba strani na zahtevo gospodarja

Register	**Opis (Hi)**	**Opis (Lo)**
39000(0x9859)	X_hi(položaj)	X_lo(položaj)
39001	Y_hi(položaj)	Y_lo(položaj)
39002	stran	število_strani
39003	rezervirano	rezervirano
39004	rezervirano	rezervirano
39005	naslov_strani	naslov_strani
39006	⋮	⋮
39024	$0x0A$	$0x0D$
meritev 1		
39025	rezervirano	rezervirano
39026	⋮	⋮
39030	podatek	podatek
39031	⋮	⋮
39049	$0x0A$	$0x0D$
Meritev 2		
⋮		
meritev 28		
39725	rezervirano	rezervirano
39726	⋮	⋮
39730	podatek	podatek
39731	⋮	⋮
39749	$0x0A$	$0x0D$

- **stran** - Pove, na kateri strani se trenutno nahajamo.
- **število_strani** - Maksimalno število strani, ki je odvisna od števila spa-rametriranih.
- **naslov_strani** - Naslov menija.
- **podatek** - Niz znakov za prikaz meritve.

3.7 Statusi digitalnih vhodov

Verjetno najlažja funkcionalnost za izvedbo je prikazovanje stanj digitalnih vhodov. Maksimalno število vhodov, ki jih lahko prikazujemo je 80, v vsakem stolpcu po 10.

Izvedba

V sklop serializacije menijev, se v globalno mapo vnese objekt, ki ima implementirano funkcijo `Get`, v kateri se osvežijo vsi statusi v registrski bazi. Statusi se neposredno preberejo iz atomične baze.

Registri protokola

Protokol podpira samo funkcijo branja in edina dinamika so statusi, ki se kličejo ciklično, kadar je prikazovalnik v meniju.

Register	**Opis (Hi)**	**Opis (Lo)**
40000(0x9859)	X_hi(položaj)	X_lo(položaj)
40001	Y_hi(položaj)	Y_lo(položaj)
40002	rezervirano	rezervirano
40003	rezervirano	rezervirano
40004	število_vhodov	rezervirano
40005	ime menija	
⋮	⋮	
40024	0	0
40029	število_di(hor)	število_di(ver)
40050	DI[7:0]	DI[15:8]
⋮	⋮	
40059	DI[71:64]	DI[79:72]

- **število_vhodov** - Število vseh DI v slave napravi.
- **število_di(hor)** - Število vrst za prikaz digitalnih vhodov.
- **število_di(ver)** - Število stolpcev za prikaz digitalnih vhodov.
- **DI[x:y]** - Stanja digitalnih vhodov.

3.8 Dogodki in alarmi

Dogodki in alarmi veljajo za pomembno funkcionalnost, saj nudijo zaščitnemu inženirju možnost učinkovitejšega nadzora, ugotavljanja in odpravljanja napak. Razlika med pojmoma dogodkov in alarmov je v njuni interpretaciji, pri čemer velja, da je vsak alarm tudi dogodek in ne velja obratno. Dogodek je vsakršna sprememba stanja v nekem trenutku, ki je lahko tudi alarm v primeru, če ga zaščitni inženir tako interpretira. Dober zgled prejšnje trditve lahko prikažem na napetostnih meritvah. Vsakršna sprememba meritve povzroči dogodek zaradi nihanja napetosti. Zaščitni inženir lahko postavi meje tolerance, kar pomeni, da če nihanje preseže postavljene meje, se dogodek interpretira kot alarm. Zahteve so mi narekovale implementacijo naslednjih funkcionalnosti:

- Serviranje navzgor omejenega seznama dogodkov, ki vključuje opis in do milisekunde natančen čas proženja dogodka.
- Serviranje liste alarmov fiksne dolžine, ki poleg opisa vsebuje še aktivnost posameznega alarma.
- Potrjevanje alarmov.

Status in potrjenost alarma - Status alarma označuje, ali je določen alarm veljaven oziroma neveljaven. Alarm dobi neveljaven status v primeru, če preseže določen pogoj, ki je po navadi vrednost ali zastavica. Specifikacija predvideva enobitne registre, ki preidejo v neveljavno stanje ob naslednjih pogojih:

- OPEN - Če se vrednost registra postavi na 0.
- CLOSED - Če se vrednost registra postavi na 1.
- CHANGE_OF_STATE - Vedno aktiven, ne glede na vrednost registra.

Vsaki na novo prispeli alarm, ne glede na status, dobi oznako nepotrjen in tako ostane, dokler jih zaščitni inženir ne potrdi. Iz uvoda je razviden pomen takšnega delovanja, zato ni potrebno dodatne razlage.

Aktivnost alarma je opredeljen glede na status, potrjenost in način delovanja, ki je določena v parametraciji. V tabeli 3.2 so prikazani vsi možni načini delovanja v relaciji z statusom in potrjenostjo alarma.

Status alarma	Potrjenost alarma	**Način delovanja**		
		Non-latched	Latched-Blinking	Latched-Steady
neaktiven	potrjen	ne sveti	ne sveti	ne sveti
aktiven	potrjen	sveti	sveti	sveti
x	nepotrjen	x	utripa	sveti

Tabela 3.2: Indikacija alarmov ali LED glede na način delovanja

Izvedba

Sprejemanje in hranjenje dogodkov skrbi funkcionalnost EventDriver, ki je izvedena na podoben način kot ostali gonilniki na aplikacijski plasti. Iz razrednega diagrama 2.2 je razvidno, da sem Modbus izvedel neposredno iz EventDriver-ja, s katerim sem si delno zagotovil dostop do željenih struktur. Kot sem opisal v razdelku načrtovanja, je veliko nevšečnosti predstavljal problem prenašanja parametrov oziroma struktur dogodkov in alarmov iz EventDriver-ja v instanco `cldu_` in obratno. Preden sem lahko serviral podatke o dogodkih in alarmih gospodarju, sem moral tako rešiti še dva problema integracije EventDriver-ja:

- serializacija,
- posredovanje dogodkov in alarmov EventDriver-ja.

Serializacija

Zamisel specifikatorja je predvidevala ločeno parametracijo alarmov in dogodkov za gonilnik EventDriver in servisni program LDU. Obstoječa implementacija EventDriver-ja je bila prosto stoječa (stand-alone) in glede na izvedenost LDU, ni dopuščala enostavne integracije v drug modul.
Serializacijo sem razdelil na inicialno in poljsko iz dveh razlogov:

- Vsiliti gonilniku EventDriver parametracijo dogodkov in alarmov. S tem sem zagotovil pravilno delovanje gonilnika in sprejem dogodkov, kljub temu, da naprava LDU ni prisotna.
- Omogočiti možnost večpoljske podpore.

V inicialni serializaciji sem gonilniku EventDriver vsilil informacije o parametriranih dogodkih in alarmih s strani uporabnika. Jasno je, da so ti podatki na voljo v serializaciji LDU, vendar EventDriver o teh ne ve nič, zato sem mu jih moral na nek način vsiliti. Za pravilno delovanje EventDriver-ja sem tako moral zagotovil naslednje podatke:

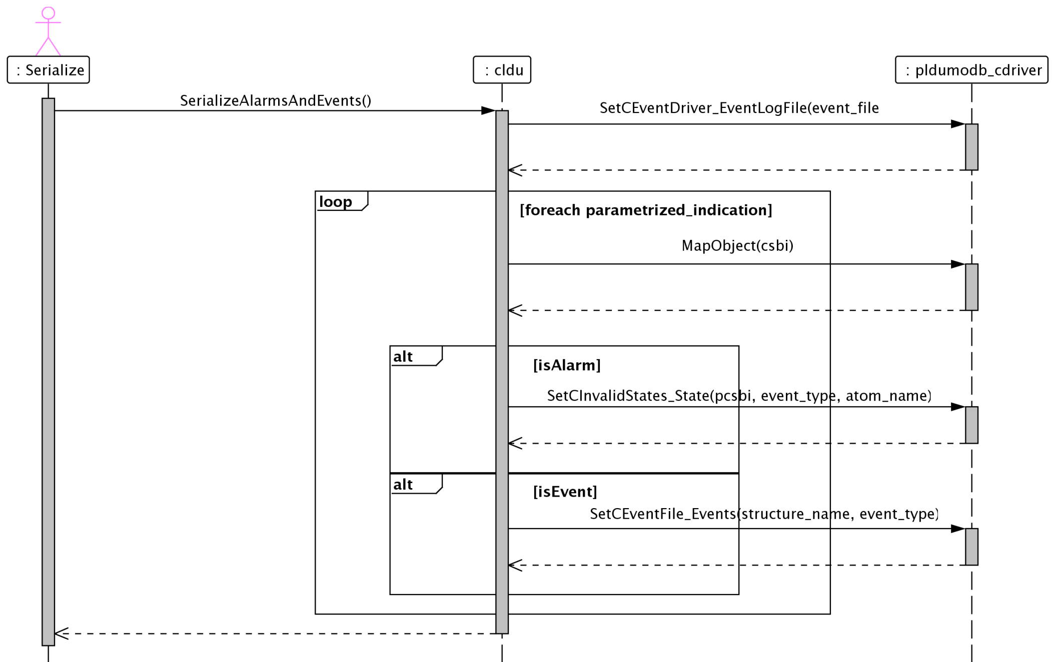
- pot do XML datoteke, kjer se hrani baza dogodkov in alarmov.
- vsiliti modulu RTDB (Real-Time DataBase) strukture, s katerim prijavim parameterirane dogodke na registre.
- vsiliti gonilniku EventDriver strukture, ki služijo kot identifikacija prejetih dogodkov.

Na sekvenčnem diagramu 3.6 je prikazan potek funkcijskih klicev v procesu inicialne serializacije alarmov in dogodkov. Iz njega je razvidno, da sem ustvaril tri funkcije, ki služijo kot posredniki med LDU-jem in EventDriver-jem. Nahajajo se v razredu `Modb::CDriver` (Modbus), ki vidi relevantne strukture iz gonilnika EventDriver. Razredni diagram 2.2 nazorno prikazuje opisano situacijo. V poljski serializaciji se strukture dogodkov in alarmov inicializirajo v takšni meri, da so pripravljene za serviranje gospodarju. Za prihodnje razvojne iteracije in večpoljsko podporo sem tako pustil odprte možnosti.

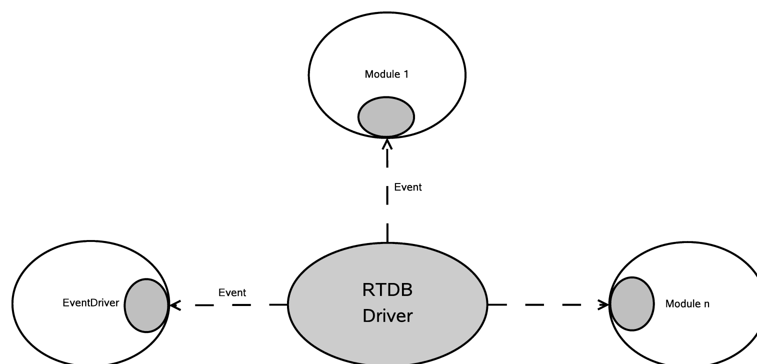
Posredovanje dogodkov in alarmov gonilnika EventDriver

Po uspešni serializaciji je EventDriver pripravljen sprejemati parametrirane dogodke. Seveda to nima nobene koristi iz perspektive LDU, saj ne obstaja mehanizem, ki bi omogočal posredovanje sprejetih dogodkov. Problem sem rešil tako, da sem v EventDriver dodal metodo `RegisterCallback`, s katero sem registriral metodo `NotifyInvalidState`, ki opravi povratek (angl. `callback`) iz LDU. Tako vsakič, ko prispe dogodek ali alarm, se posreduje v LDU prek registrirane funkcije.

LDU sprejme dogodek ali alarm v funkciji `NotifyInvalidState` in ga vpiše v strukturi `CEvents` oziroma `std::set`. Število dogodkov za prikaz je omejeno in v primeru prekoračitve se najstarejši prepišejo z novimi. Hranijo se v vektorju, ki ga obkroža (angl. wrap) struktura `CEvents` z namenom pravilnega



Slika 3.6: Serializacija dogodka ali alarma



Slika 3.7: Distribucija dogodkov

dodajanja novih dogodkov. Sedaj so dogodki in alarmi na voljo za serviranje gospodarju. Celoten potek sprejema enega dogodka ali alarma je prikazan na sekvenčnem diagramu v dodatku A.2.

Koda 5 - Registracija “callback“ funkcije za sprejem dogodkov

```
void CEventDriver::RegisterCallback(void(*pf)(SEvent*));

/* Registracija funkcije v Modb::CDriver */
RegisterCallback(pldu_->NotifyInvalidState);
```

Protokol - prikaz dogodkov

Dogodki in alarmi se prikazujejo v ločenih menijih in posledično imajo tudi ločen protokol. V protokol dogodkov sta vključeni funkciji branja in pisanja. Gospodar piše v register v dveh primerih:

- v zahtevi po spremembi strani
- postavi zastavico 0 v register, ki indicira prihod novega dogodka.

Za dogodke je tako predvidenih 14 strani po 7 dogodkov, vse skupaj 98 dogodkov. Če se preseže maksimalno število dogodkov, se starejši začnejo prepisovati. Za vsak dogodek sta predvideni po dve vrstici.

Register	**Opis (Hi)**	**Opis (Lo)**
30000(0x7530)	X_hi(položaj)	X_lo(položaj)
30001	Y_hi(položaj)	Y_lo(položaj)
30002	stran	število strani
30003	rezervirano	nov_dogodek
30004	rezervirano	rezervirano
30005	**Ime menija**	
⋮	0x0A	0x0D
30024	0	0
30025	**Dogodek 1**(na izbrani strani)	
⋮	⋮	
30030	podatek(prva vrstica)	
⋮	⋮	
30051	0	0
30052	podatek(druga vrstica)	
⋮	⋮	
30074	0	0
30325	**Dogodek 7**(na izbrani strani)	
⋮	⋮	
30374	0	0

- **stran** - Trenutna stran prikaza dogodkov.
- **število_strani** - Število strani, glede na število dogodkov.
- **naslov_strani** - Naslov menija.
- **nov_dogodek** - Indikacija gospodarju o novem prispelem dogodku in velja le za prvo stran dogodkov.
- **podatek(prva vrstica)** - Vsebuje opis dogodka.
- **podatek(prva vrstica)** - Vsebuje do milisekunde natančen čas dogodka.

Protokol - prikaz in potrjevanje alarmov

V protokol sta vključeni funkciji branja in pisanja. Ko uporabnik preide na določeno stran alarmov, se ti preberejo le enkrat, kajti lista imen je statična.

Dinamični so statusi alarmov na aktualni strani, ki jih gospodar ciklično kliče. Podobno kot pri analognih meritvah in dogodkih, gospodar piše v register na zahtevo spremembe strani ali potrjevanje alarmov.
Maksimalno število alarmov na stran je 14, pri čemer je na voljo 7 strani.

Statusi alarmov		
Register	**Opis (Hi)**	**Opis (Lo)**
29900(0x74CC)	alarm 7-4	alarm 3-0
29901	alarm 15-12	alarm 11-8
⋮	⋮	⋮
29924	alarm 199-196	alarm 195-192
29925	rezervirano	rezervirano

Potrjevanje strani alarmov					
Register	**Opis**				
	bit 15	bit 14	bit 13	...	bit 0
29926(0x74E6)	rezervirano	rezervirano	potrditev alarma 14	...	potrditev alarma 1

Za vsak alarm sta namenjen po dva bita, ki označujeta status glede na način delovanja.

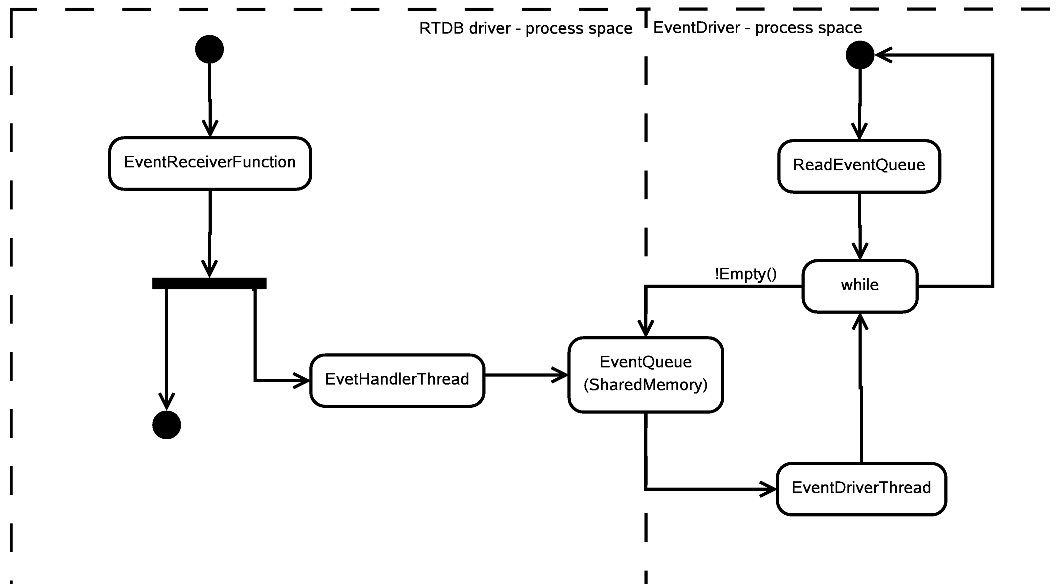
Vsebina alarmne liste		
Register	**Opis (Hi)**	**Opis (Lo)**
22000(0x55F0)	rezervirano	rezervirano
22001	rezervirano	rezervirano
22002	stran	število strani
22003	rezervirano	rezervirano
22004	rezervirano	rezervirano
22005	**Ime menija**	
⋮	0x0A	0x0D
20024	0	0
20025	**Alarm 1**	
⋮	⋮	
22030	podatek	
⋮	0x0A	0x0D
22049	0	0
22350	**Alarm 14**	
⋮	⋮	
22355	podatek	
⋮	0x0A	0x0D
22374	0	0

- **stran** - Trenutna stran prikaza alarmov.
- **število_strani** - Število strani alarmov.
- **podatek** - Tekst, ki opisuje alarm.

3.8.1 Predelava načina sprejema dogodkov

Za distribucijo dogodkov modulom, ki se prijavijo za sprejemanje, skrbi gonilnik RTDB. V zaključni fazi testiranj so se izvajali vztrajnosti testi, ki so odkrili "memory leak" v gonilniku RTDB. Del kode vsakega prijavljenega modula, se izvaja v procesnem prostoru gonilnika RTDB, zato je bilo iskanje dejanskega krivca za napako močno oteženo. Izkazalo se je, da ga je manifestiral sistem za sprejem dogodkov v EventDriver-ju ob močnih obremenitvah.

Vsak prispeli dogodek se obravnava v novi kratkotrajni niti, ki doda dogodek na sklad v skupni pomnilnik (angl. shared memory - SHM). Očitno ima Kernel 2.4.9 težave z nitmi v primeru, če je le teh ogromno. Problem sem



Slika 3.8: Sprejem dogodkov z IPC ”SharedMemory”

rešil z drugačnim načinom medprocesne komunikacije, imenovane cevi (angl. named pipes). Pri tem sem izkoristil možnost, ki mi ga ponuja rcman, saj ima EventDriver že po privzetem ustvarjeno cev, iz katere dobiva sporočila iz nižje ležečih plasteh. Tako sem moral RTDB modulu povedati lokacijo cevi, oblikovati, poslati ter sprejeti sporočilo.
Za lokacijo cevi sem oblikoval posebno strukturo `SharedProperties`, ki med drugimi vsebuje edinstven ID modula. Kazalec na strukturo posredujem funkciji `AddEventNotification`, s katero se prijavim za sprejemanje dogodkov. Tako vsak sprejeti dogodek v funkciji `SOE_EVENTS_NOTIFICATION_FUNCTION`, dobi kazalec na strukturo `SharedProperties`, iz katere lahko sestavim pot do cevi. Cevi medprocesne komunikacije v rcman-u imajo standardna imena v obliki:

```
<pipe\_path> <module\_1\_id> <comm\_direction(<-|->)> <module\_2\_id>
```

V mojem primeru oblikujem ime cevi ”$4 < -4$“, na katerega RTDB pošilja dogodke EvetnDriver-ju. Sprejem dogodkov na cevi poskrbi rcman, ki jih dostavi funkciji `OnTransmitMessage` ali `OnReceivedMessage`, odvisno od modela komunikacije (slave/master). Sporočila se prenašajo po cevi v seriji bitov, zato sem izbral najlažjo rešitev in vse informacije o dogodku spravil v strukturo `Event`. Pomembno je, da v strukturi ni nobenih kazalcev, sicer je potrebno strukturo posebej serializirati, kar pomeni dodatne probleme.

Koda 6 - Event struktura ustrezna za pošiljanje v seriji bajtov

```
struct Event{
  int iAddress;
  bool bEventHronology;
  unsigned char chStructureType;
  DBSTRUCTURES_FLAGS0 usFlags;
  DBSTRUCTURES_StructureValue structureValue;
  timeval timestamp;
  char chStringValue[MAX_STRING_LEN];
};
```

3.9 Uporabniško nastavljive diode LED

Podobno uporabniško namembnost kot alarmi, imajo tudi uporabniško nastavljive diode LED. Na voljo je osem LED, ki v primeru neveljavni stanj, ustrezno svetijo ali utripajo glede na parametracijo. Na eno LED je možno vezati več različnih registrov oziroma signalov, zato med njimi veljajo določena pravila:

- velja disjunkcija signalov, če je kateri od njih aktiven.
- če je aktivnih več signalov, ima prednost tisti z najvišjo prioriteto. Prioritete so določene glede na tabelo 3.2, kjer je NON-LATCHED najnižja in LATCHED-BLINKING najvišja prioriteta.
- vsak signal vezan na diodo mora biti tudi alarm.

V skladu z opisanimi pravili je moja naloga implementacija serviranja ustreznih stanj LED gospodarju, s pomočjo struktur, ki jih nudi EventDriver. Gonilnik mora tudi omogočati potrjevanje vseh LED.

Izvedba

Velika večina dela za izvedbo serviranja podatkov o stanjih LED, sem opravil z integracijo gonilnika EventDriver v prejšnjem razdelku. Bazo podatkov o LED na žalost ne hrani EventDriver, zato sem moral to opraviti v LDU.

V gonilnik sem dodal strukturo `LED`, ki hrani vse potrebne podatke za oblikovanje ustrezne indikacije za posamezno LED. Ker imam opravka s konstantnim številom LED, sem ustvaril seznam osmih vektorjev, ki vsebuje omenjeno strukturo.

Koda 7 - Hranjenje relevantnih podatkov za realizacijo stanj diod LED

```
struct LED{
  LED() :                 not_acknowledged(false) {};
  LatchingMode            latching;
  // Ime signala v atomični bazi in služi kot enoviti ID.
  std::string             structure_name;
  // Indicira potrjenost diode
  bool                    not_acknowledged;
};

std::vector<LED *> leds[8];
```

V procesu serializacije dogodkov in alarmov, se za vsako diodo v seznamu inicializirajo podatki o stanjih. V konstruktorju sem poskrbel, da so na začetku vse diode potrjene. Ime strukture mora biti enovita, ker služi za identifikacijo prispelega alarma.

Stanje diod se spreminja v skladu s statusom alarma (veljaven/neveljaven) in njegove potrjenosti. Slednje potrjuje uporabnik na prikazovalni enoti, pri čemer gospodar zahteva funkcijo pisanja. V tem sklopu se vse zastavice postavijo na false.

Protokol

V protokol je vključena tako funkcija branja kot pisanja. Funkcijo pisanja gospodar zahteva v primeru potrjevanja diod, kjer v register vpiše vrednost 0. Gospodar registre bere ciklično, pri čemer na vsako zahtevo LDU posodobi registrsko bazo. Za vsako diodo sta rezervirana po dva bita in glede na tabelo 3.2, se v registrsko bazo vpisujejo naslednje vrednosti:

- 0x00 - LED ne sveti
- 0x01 - LED utripa
- 0x11 - LED sveti

Register	Opis (Hi)	Opis (Lo)
500(0x01F4)	LED 7-4	LED 3-0

Poglavje 4

Sklepne ugotovitve

V prejšnjih poglavjih sem obravnaval razvoj servisnega programa, ki se je razprostiral skozi tri razvojne iteracije. Nastala zasnova iz prve iteracije se praktično ni spreminjala, temveč se je samo nadgrajevala. V zadnji iteraciji so prišle funkcionalnosti, ki so zahtevale integracijo gonilnika EventDriver in pokazale so se slabosti prvotne zasnove. Nastal je problem slabo definiranega vmesnika oziroma enkapsulacije. S tem je močno otežen nadaljnji razvoj gonilnika ter iskanje napak oziroma hroščev. Na račun slednjih sem izgubil veliko časa, ker obstoječi vmesnik zahteva veliko pomnjenja o celotnem delovanja obstoječe izvorne kode. Pri manjših programih to ne predstavlja tako velikega problema, v nasprotnem primeru pa je zelo težko držati na znanju vse trike in rešitve za nastale probleme.

Ker je gonilnik LDU zgolj uporabniška aplikacija in se močno opira na ostale module v sistemu, sem potreboval ogromno časa za prebiranje tuje programske kode. Na voljo je bila slaba oziroma v nekaterih primerih celo nikakršna dokumentacija, zato mi je bilo delo močno oteženo, še posebej zaradi dejstva, ker nisem bil izvedenec na področju zaščit v elektroenergetiki. Določene informacije sem tako moral iskati pri avtorjih zadevnih modulov, ki mi včasih tudi niso znali podati jasnih odgovorov. Dokumentacijo programske opreme, tako velikih kot kot majhnih projektov, enostavno ne smemo zanemarjati. Na tak način lahko privarčujemo veliko časa in truda za nadaljnji razvoj ter posledično tudi denarja.

Kljub vsem problemom sem skozi testiranje gonilnika ugotovil, da implementirane funkcionalnosti delujejo v solidnem performančnem območju. Poraba pomnilniških resursov je glede na obseg gonilnika normalna in še pomembneje stabilna. Veliko prizadevanj sem vložil ravno v zagotavljanje stabilnosti gonilnika. Zaradi neizkušenosti v programskem jeziku C++ sem nevede zane-

marjal problem sprostitve pomnilniških resursov. Odkrivanje tovrstnih napak je bilo težavno, ker pogosto ni obstajal očiten krivec. To se je še posebej izkazalo v sistemu za sprejemanje dogodkov EventDriver-ja, pri čemer smo za lociranje in odpravo napake potrebovali tri tedne in po metodi eliminacije ugotovili najverjetnejšega krivca. Seveda so pri odkrivanju napake prisostvovali tudi drugi razvijalci na ostalih potencialnih krivcih.

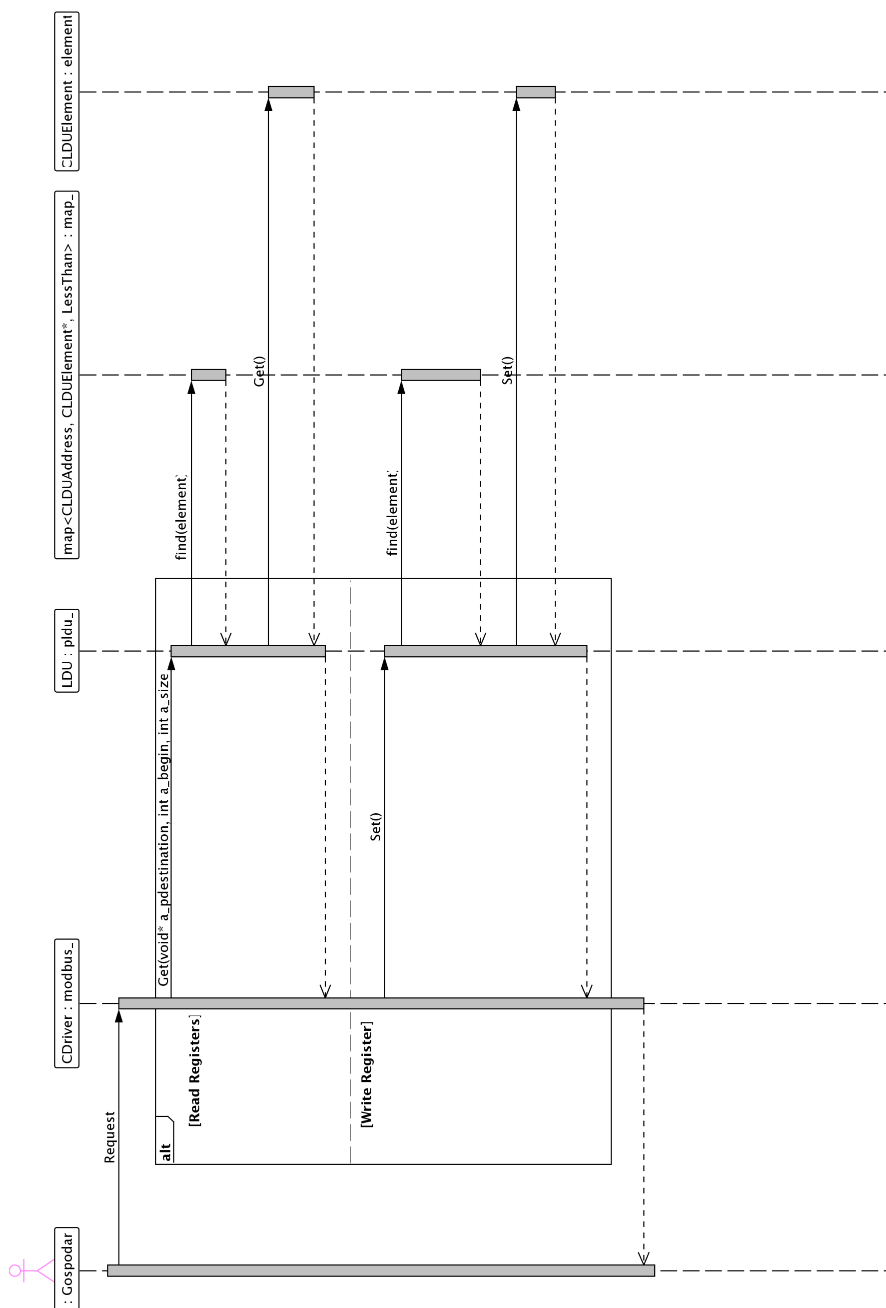
Skozi razvoj gonilnika, še posebej v fazi tranzicije, sem izkusil nevšečnosti, ki doletijo predvsem razvijalce uporabniških vmesnikov. Vsakršna napaka, ki jo odkrije tester ali končni uporabnik, se takoj okrivi razvijalca vmesnika, kljub temu da ni dejanski povzročitelj. Določen delež hroščev je bilo tovrstnih, vendar jih je večina vendarle nastala zaradi površnosti, saj velja pregovor *"motiti se je človeško"*.

Izvorna koda servisnega programa zaradi obsežnosti ni vključena v samo diplomsko delo, je pa priložena v elektronski obliki.

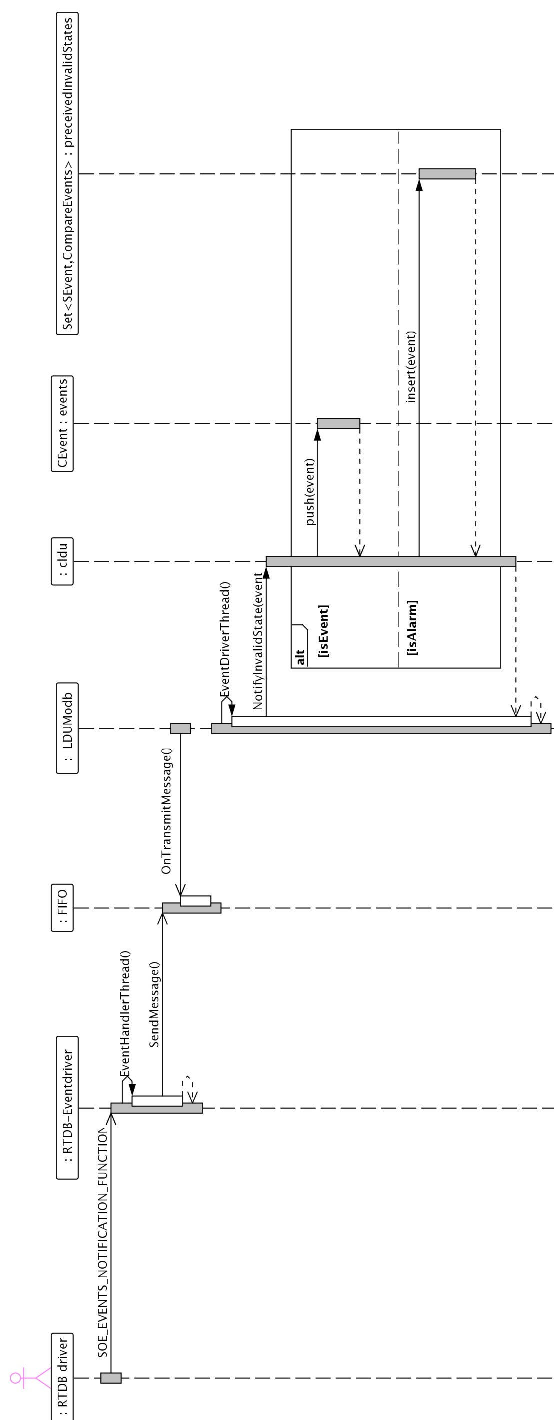
Dodatek A

Sekvenčna diagrama komunikacije

Na sliki A.1 je predstavljen sekvenčni diagram branja ali pisanja v registrsko bazo (kot je definirana s protokolom Modbus) servisnega programa, na sliki A.2 pa je predstavljen sekvenčni diagram komunikacije servisnega programa s sistemom rcman.



Slika A.1: Sekvenčni diagram branja ali pisanja v registrsko bazo servisnega programa.



Slika A.2: Sekvenčni diagram komunikacije servisnega programa s sistemom rcman.

Slike

1.1 Sekundarni elektroenergetski sistem 4
1.2 Funkcionalnost LDU 5

2.1 Komunikacija Modbus 11
2.2 Hierarhija razredov za izvedbo protokola Modbus 14
2.3 Sekvenčni diagram dekodiranja sporočila 16
2.4 RS-232 zanka s pomoč dveh USB pretvornikov 17

3.1 Osnovni meni na prikazovalniku 24
3.2 Binarna slika 25
3.3 Primer animiranih okvirov ločilke v različnih stanjih 26
3.4 Serializacija analognih meritev 33
3.5 Sprememba strani na zahtevo gospodarja 34
3.6 Serializacija dogodka ali alarma 39
3.7 Distribucija dogodkov 39
3.8 Sprejem dogodkov z IPC "SharedMemory" 44

A.1 Sekvenčni diagram branja ali pisanja v registrsko bazo servisnega programa. 50
A.2 Sekvenčni diagram komunikacije servisnega programa s sistemom rcman. 51

Tabele

2.1 Splošna struktura telegrama 8
2.2 Vprašanje gospodarja o registrih 9
2.3 Odgovor, ki vrne vse zahtevane registre 10
2.4 Vprašanje in odgovor pri pisanju v register 11
2.5 Arhitektura rcman 13

3.1 Statusi animiranih elementov 28
3.2 Indikacija alarmov ali LED glede na način delovanja 37

Literatura

[1] Martin Fowler, "A Brief Guide to the Standard Object Modeling Language", *UML Distilled Third Edition*, Addison-Wesley, 2003

[2] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, "Elements of Reusable Object-Oriented Software", *Design Patterns*, Addison-Wesley, 1994

[3] Marc J. Rochkind, *Advanced UNIX programming*, 2nd edition, Addison-Wesley, 2004

[4] Stephen Prata, *C++ Primer Plus*, 5th edition, Sams, 2005

[5] IEEE (2002) Zapiski predavanj,
Dostopno na: http://lees.fe.uni-lj.si/www/Images/File/Zascita/Zascita_pogl1-3.pdf

[6] Modbus,
Dostopno na: http://www.lammertbies.nl/comm/info/Modbus.html