TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

TOMMI REKOSUO

NÄYTTÖLASIN KONTROLLIYKSIKKÖ

Diplomityö

Tarkastaja: Prof. Hannu-Matti Järvinen
Tarkastaja ja aihe hyväksytty
Tieto- ja sähkötekniikan tiedekuntaneuvoston kokouksessa 4. marraskuuta 2015

TIIVISTELMÄ

TOMMI REKOSUO: Näyttölasin kontrolliyksikkö
Tampereen teknillinen yliopisto
Diplomityö, 53 sivua, 22 liitesivua
Joulukuu 2015
Signaalinkäsittelyn ja tietoliikennetekniikan koulutusohjelma
Pääaine: Sulautetut järjestelmät
Tarkastajat: Prof. Hannu-Matti Järvinen
Avainsanat: can, j1939, ohjelmisto, säie, näyttö, yaml, läpinäkyvä

Tässä diplomityössä toteutettiin ohjelmisto osana uuden läpinäkyvän näyttötekniikan prototyyppiä. Työn tilaaja, Pilkington Automotive Oy, asensi Beneq Products Oy:n valmistamat näytöt ajoneuvon tuulilasiin ja TTY:n tehtävänä oli toteuttaa näyttöjä ohjaava ohjelmisto.

Ohjelmisto kuuntelee ajoneuvon CAN-väylältä J1939-protokollan mukaisia viestejä ja Tampereen yliopiston toteuttamaa käyttöliittymää TCP-yhteyden kautta. Vastaanotettujen viestien ja ohjauksen perusteella valitaan, mitä näytöllä näytetään. Työssä tarkastellaan ohjelmiston mahdollisia arkkitehtuureja ja toteutustapoja.

Arkkitehtuurin valinnassa päädyttiin ratkaisuun, jossa CAN-väylän ja käyttöliittymän rajapinnoille on omat säikeet. Kolmas säie sisältää toimintalogiikan ja ohjaa viestien perusteella näyttöyksikköä. Toteutuksessa päädyttiin käyttämään konfiguraatiotiedostoja joiden avulla erilaisten näyttöjen käyttö on mahdollista. Lisäksi CAN-väylältä luettavat viestit valitaan konfiguraatiotiedoston perusteella.

ABSTRACT

TOMMI REKOSUO: Control Unit of Glassdisplay
Tampere University of Technology
Diplomityö, 53 pages, 22 Appendix pages
December 2015
Master's Degree Programme in Signal Processing and Telecommunications Technology
Major: Embedded Systems
Examiner: Prof. Hannu-Matti Järvinen
Keywords: can, j1939, program, thread, display, yaml, transparent

In this master's thesis a software was developed as a part of a new prototype of a new transparent display technique. The client in this project was Pilkington Automotive Oy, which installed the screens manufactured by Beneq Products Oy to a vehicle's windshield. Tampere University of Technology's task was to develop software that controls the displays.

The software listens to the vehicle's CAN bus which uses the J1939 protocol. The user interface which was developed by University of Tampere is also connected to the software by a TCP connection. The software decides what to display based on the messages received from these two interfaces. This thesis discusses the different architectures and approaches for the software.

The chosen architecture uses three threads: one for the CAN bus and one for the user interface. A third thread listens received messages from the other two threads and controls the display engine based on them. The software uses configuration files which makes it possible to use different displays. In addition the messages from CAN bus are filtered based on a configuration file.

ALKUSANAT

Tämä diplomityö on tehty Tampereen teknillisen yliopiston Tietotekniikan laitoksella. Työ on osa Pilkington Automotive Oy:lle toteutettua prototyyppiä. Työn ohjaajana toimi Hannu-Matti Järvinen, jota haluan kiittää työn tarkastuksesta ja hyvästä ohjauksesta. Lisäksi kiitän perhettäni ja ystäviäni, jotka ovat tukeneet minua opiskelujeni aikana. Haluan myös kiittää samassa projektissa työskennellyttä Miikka Juomojaa hyvästä yhteistyöstä sekä työssä auttaneita TTY:n henkilökuntaan kuuluvia.

Tampereella, 13.11.2015

Tommi Rekosuo

SISÄLLYS

1. Johdanto 1

2. Vaatimukset 3

2.1 Yleisiä vaatimuksia 3

2.2 Dokumentointi 3

2.3 Rajapinnat 4

2.4 Laitteisto 4

2.5 Ohjelmisto 5

3. Käytössä oleva ympäristö ja tekniikka 6

3.1 CAN 6

3.1.1 Tiedonsiirto 6

3.1.2 Fyysinen rakenne 8

3.2 J1939 9

3.2.1 Kehysrakenne 9

3.2.2 Viestit ja osoitteistus 10

3.2.3 Linux-tuki 11

3.3 Laitteisto 11

3.4 Kehitysympäristö 12

3.5 YAML-konfiguraatio 13

3.6 Näyttölasi 14

4. Ratkaisuvaihtoehdot 18

4.1 Ajuri 18

4.2 Arkkitehtuuri 19

4.2.1 Yksi ohjelma 19

4.2.2 Erilliset ohjelmat 20

4.3 Yksi vai useampi säie 21

4.4 Konfiguroitavuus 21

4.5 Valittu toteutus 22

5. Toteutus 24
5.1 Konfigurointi 24
5.1.1 Näyttöjen konfigurointi 24
5.1.2 CAN-kuuntelijan konfigurointi 25
5.1.3 UI-kuuntelijan konfigurointi 26
5.2 Rajapinnat 26
5.2.1 Näyttöyksikkö 27
5.2.2 Käyttöliittymä 27
5.2.3 CAN-väylä 28
5.2.4 Sisäiset rajapinnat 29
5.3 Ohjelmiston rakenne 31
5.3.1 CAN-väylän kuuntelija 31
5.3.2 UI-kuuntelija 33
5.3.3 Pääsäie 35
5.3.4 Konfiguraation luku 36
5.3.5 Tukitoiminnot 37
5.4 Testaus 38
5.4.1 Rajapinnat 38
5.4.2 Ohjelmisto 40
5.4.3 CAN-nauhoite 41
5.4.4 Järjestelmätestaus 41
5.5 Dokumentaatio 42
6. Tulokset ja niiden tarkastelu 43
6.1 Prototyyppi 43
6.2 Jatkokehitys 43
6.2.1 SPI-väylän korvaaminen 44
6.2.2 Ohjelmisto 45
6.2.3 Rajapinnat 46
6.2.4 Kehitysympäristö 48
6.2.5 Elektroniikka 48

7. Yhteenveto 49

Lähteet 50

LIITE 1. YAML-konfiguraatio

LIITE 2. Käyttöliittymän protokolla

LIITE 3. Näyttöyksikön rajapinta

KUVALUETTELO

1.1 Projektin osa-alueet. 2

3.1 Tavallinen CAN-kehys. 8

3.2 CAN-kehys 29-bittisellä tunnisteella. 8

3.3 J1939-kehyksen alkuosa. 10

3.4 MYIR kehitysalusta. 12

3.5 Kehitysympäristön rakenne. 13

3.6 Ensimmäinen näyttölasi. 15

3.7 Toinen näyttölasi. 16

3.8 Kolmas näyttölasi. 16

3.9 Neljäs näyttölasi. 17

4.1 Valitun toteutuksen arkkitehtuuri. 23

5.1 CAN-kuuntelijan toimintaperiaate. 32

5.2 UI-kuuntelijan toimintaperiaate. 34

5.3 Kontrolliyksikön toimintaperiaate. 36

5.4 Testaustyökalun käyttöliittymä. 39

LYHENTEET JA MERKINNÄT

ARM	Acorn RISC Machine. RISC-mikrokontrolleriarkkitehtuuri, jota käytetään pienissä laitteissa.
AWG	American Wire Gauge. Johdinpaksuutta ilmaiseva standardi.
Bluetooth	Langaton tiedonsiirtotekniikka lyhyille etäisyyksille.
C	Ohjelmointikieli, jota käytetään usein sulautetuissa järjestelmissä.
CAN	Controller Area Network. Tiedonsiirtoväylä, jota käytetään paljon autoteollisuudessa.
Code::Blocks	Ohjelmiston kehitysympäristö.
CSMA	Carrier Sense Multiple Access. Tiedonsiirtoväylän varausmenetelmä, jossa lähettävät laitteet kuuntelevat väylää huomatakseen konfliktin.
Debug	Ohjelmiston toiminnan tarkkailu, virheiden etsiminen ja korjaaminen.
DM1	J1939-protokollan diagnostiikkaviesti, jossa ilmoitetaan aktiiviset viat.
Doxygen	Dokumentointityökalu, joka tuottaa dokumentaation ohjelmakoodin kommenttien perusteella.
Eclipse	Ohjelmiston kehitysympäristö.
EmIDE	Ohjelmiston kehitysympäristö.
Ethernet	Yleisesti käytössä oleva tietoliikennetekniikka.
FIFO	First In First Out. Puskuri, jossa ensimmäisenä sisään tullut alkio poistuu ensimmäisenä.
GDB	The GNU Debugger. Debug-työkalu ohjelmille.
GPIO	General Purpose Input/Output. Yleiskäyttöinen pinni, jota voidaan kuunnella tai ohjata.
HDMI	High-Definition Multimedia Interface. Äänen ja kuvan siirtoon tarkoitettu liitäntästandardi.
IDE	Integrated Development Environment. Ohjelmistojen kehitystyökalu, joka sisältää hyödyllisiä oheistoimintoja.
Infix	Laskutoimituksen esitysmuoto, jossa laskuoperaatiot ovat operandien välissä.
IOCTL	Input/Output Control. Järjestelmäkutsu, jolla ohjataan laitteita.
IP	Internet Protocol. Tietoliikenneväylissä käytetty tiedonsiirtoprotokolla.
ISO	International Organization for Standardization. Standardointijärjestö.

J1939	CAN-väylän protokolla, jota käytetään raskaissa ajoneuvoissa, kuten työkoneissa ja maatalouskoneissa.
JSON	JavaScript Object Notation. Tiedostomuoto, jolla voidaan kuvata dataobjekteja.
Leap Motion	Käden liikkeitä tunnistava sensori, joka hyödyntää infrapunavaloa.
Libyaml	C-kielellä toteutettu kirjasto YAML-tiedostojen lukuun.
MilCAN	Sotilasajoneuvoissa käytetty CAN-väylän protokolla.
Mutex	Mutual Exclusion. Käytetään rinnakkaisessa ohjelmoinnissa resurssin lukitsemiseen yhdelle säikeelle.
NMEA 2000	Merenkulussa käytetty CAN-väylän protokolla.
NRZ	Non-return-to-zero. Tiedonsiirron koodausmenetelmä, jossa ei ole neutraalia tilaa vaan väylä on joko 1 tai 0.
OLED	Organic Light-Emitting Diode. Näyttötekniikka, jossa valoa säteilevä kerros on orgaanista yhdistettä.
OSI-malli	Open Systems Interconnection Reference Model. 7-kerroksinen malli, joka kuvaa tiedonsiirtoprotokollien suhdetta.
Perl	Korkean tason ohjelmointikieli.
PGN	Parameter Group Number. J1939-protokollan viestin tunnisteena toimiva parametriryhmä.
Postfix	Laskutoimituksen esitysmuoto, jossa laskuoperaatiot kirjoitetaan operandien jälkeen.
Qt	Ohjelmistojen ja graafisten käyttöliittymien kehitysympäristö.
QtCreator	Qt:n kehitystyökalu.
Ristikäännös	Ohjelmisto käännetään eri alustalla kuin millä se on tarkoitus ajaa.
SAE	Society Of Automotive Engineers. Autoteollisuuden standardointijärjestö.
Shunting-yard	Dijkstran kehittämä algoritmi, jolla infix-notaatio saadaan postfix-muotoon.
SPI	Serial Periphreal Interface. Synkronoitu sarjamuotoinen tietoliikenneväylä.
SPN	Suspect Parameter Number. J1939-protokollassa parametrin tunniste.
SSH	Secure Shell. Tiedonsiirtoprokotolla, jossa on vahva salaus.
SVN	Subversion. Versionhallintajärjestelmä.
TASEL	Transparent Electroluminescent. Elektroluminesenssia hyödyntävä näyttötyyppi, jossa tausta on läpinäkyvä.
TCP	Transmission Control Protocol. Yhteydellinen tiedonsiirtoprotokolla, joka sisältää virhetarkistuksen.
UDP	User Datagram Protocol. Yhteydetön tiedonsiirtoprotokolla.

UI	User Interface. Käyttöliittymä.
Valgrind	Työkalu, jolla voidaan tarkkailla ohjelman muistinkäyttöä ja etsiä esimerkiksi muistivuotoja.
XML	Extensible Markup Language. Yleisesti käytössä oleva merkintäkieli.
YAML	YAML Ain't Markup Language. Merkintäkieli, joka on suunniteltu ihmisystävälliseksi.

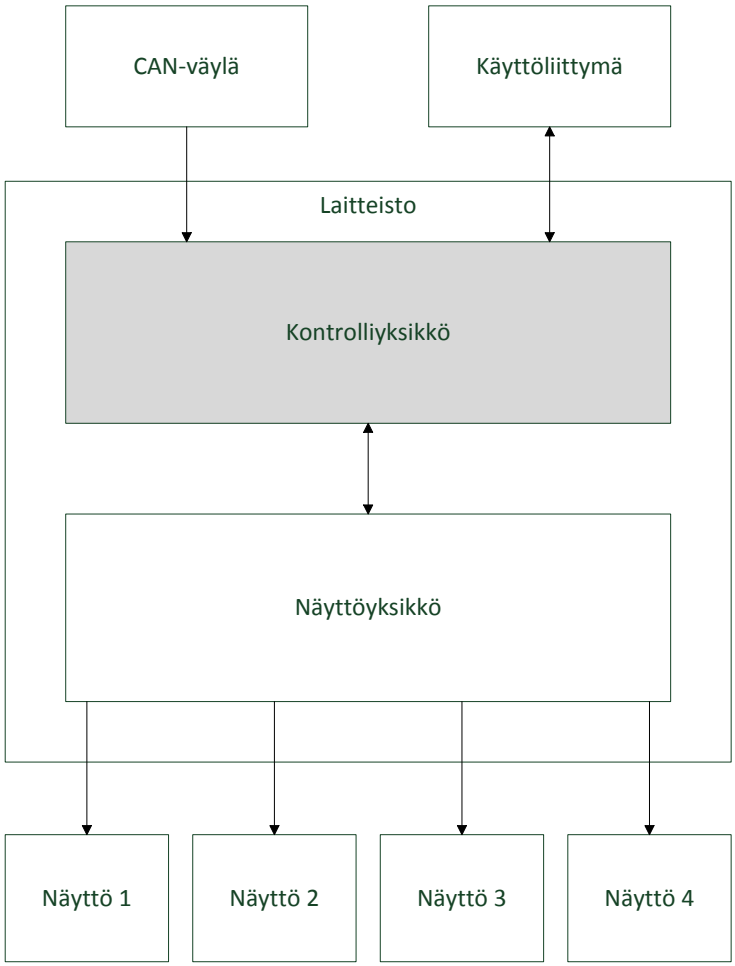
1. JOHDANTO

Visuaalinen tiedonvälitys on olennainen osa monien laitteiden käyttöliittymää ja pääsääntöisesti tähän tarkoitukseen käytetään erilaisia näyttölaitteita. Välitettävä tieto vaihtelee käyttötarkoituksen mukaan, mutta yleensä ainakin osa tiedosta liittyy jollain tavalla laitteen toimintaan. Sulautetuissa järjestelmissä tämä näkyy selkeästi. Esimerkiksi televisiossa varsinaisen lähetyksen lisäksi voidaan tarkastella näytön asetuksia tai astianpesukoneesta nähdään mikä pesuvaihe on menossa. Se, missä muodossa ja miten tietoa näytetään, on jatkuvan kehityksen alla. Nykyisin lähes jokaisessa laitteessa on jotain tietotekniikkaa, joka kerää tietoa laitteen toiminnasta, jolloin käyttäjälle voidaan näyttää yhä enemmän yksityiskohtaisempaa tietoa.

Tämä diplomityö on osa Pilkington Automotive Finland Oy:lle toteutettua prototyyppiä, jossa käytettiin Beneq Oy:n valmistamia läpinäkyviä TASEL-näyttöjä (engl. Transparent Electroluminescent Display) [4]. Näytöt hyödyntävät elektroluminesenssi-ilmiötä, jossa materiaalin läpi johdettu sähkövirta saa sen tuottamaan sähkömagneettista säteilyä näkyvän valon alueella. Elektroluminesenssiin perustuvat näytöt eivät ole uusi keksintö, mutta sen sijaan TASEL-näytöt ovat siinä mielessä uutta teknologiaa, että lasiin on kiinteästi sijoitettu ledien tapaan valaisevia segmenttejä, jotka pois päältä ollessaan ovat läpinäkyviä. Läpinäkyvyys mahdollistaa näyttöjen sijoittamisen esimerkiksi ajoneuvon tuulilasiin tai muuhun sellaiseen paikkaan, jossa myös taustan näkyminen on tärkeää. Esimerkiksi monenlaiset infonäytöt voisivat olla TASEL-näyttöjen käyttökohteita.

Pilkington Automotive Oy:n motiivina käyttää tätä uutta teknologiaa on tarjota ajoneuvojen tuulilaseille näyttöjen myötä lisäarvoa, joka taas on suuri etu kilpailijoihin nähden. Perinteisten lasien valmistus on kuitenkin hyvin kilpailtua. Tästä syystä Pilkington onkin kehittänyt toimintaansa tarjoamalla vaikeammin valmistettavissa olevia erikoislaseja moneen eri käyttöön ja näyttöjen lisääminen laseihin on osa tätä kehitysprosessia.

Pilkington asensi näytöt traktorin tuulilasiin. Tampereen teknillisen yliopiston tehtävä oli toteuttaa näytölle ajuri sekä ohjelmisto, joka kuuntelee traktorin CAN-väylää (engl. Controller Area Network) [7] ja käyttöliittymältä tulevia viestejä.



Kuva 1.1 *Projektin osa-alueet. Tummennetut kohdat on toteutettu tässä diplomityössä.*

Käyttöliittymän suunnittelusta ja toteutuksesta vastasi Tampereen yliopisto. Tässä diplomityössä toteutettiin CAN-väylää ja käyttöliittymää kuunteleva kontrolliyksikkö, joka ohjaa saatujen viestien perusteella näyttöjä lähettämällä viestejä toisessa diplomityössä tehtyyn näyttöä ohjaavaan ohjelmistoon. Tarkoituksena on löytää sellainen ohjelmistoarkkitehtuuri, joka sopii tähän tarkoitukseen ja toteuttaa se. Kuvassa 1.1 on esitetty projektin osa-alueet yleisellä tasolla, joista tummennettuna tämän työn osuus. Projektin tavoitteena on toteuttaa prototyyppi, joka esitellään messuilla.

Työn tarkemmat vaatimukset esitellään luvussa 2. Luvussa 3 tarkastellaan käytössä olevaa ympäristöä ja tekniikkaa, jonka jälkeen luvussa 4 esitellään vaatimusten ja ympäristön pohjalta erilaisia ratkaisuvaihtoehtoja. Luvussa 5 kuvataan toteutettu ohjelmisto ja luvussa 6 esitellään tulokset ja niiden tarkastelu.

2. VAATIMUKSET

Tässä luvussa esitellään, mitä ollaan tekemässä ja mitä vaatimuksia työlle on asetettu. Projektissa toteutettava prototyyppi oli ensimmäinen laatuaan, joten joitakin tekniikan asettamia rajoituksia tuli ilmi myös projektin edetessä.

2.1 Yleisiä vaatimuksia

Prototyyppi toteutettiin yhteistyössä Tampereen yliopiston kanssa. TaY:n tehtävänä oli käyttöliittymän toteutus. Käyttöliittymä toimii erillisessä laitteessa ja käskyt välitetään TTY:llä toteutetulle näyttölaitteen ohjelmistolle. Olisi myös saattanut olla mahdollista, että käyttöliittymäohjelmisto ajetaan samalla laitteistolla, mutta niiden erillään pitäminen tässä tapauksessa jättää enemmän valinnanvaraa käyttöliittymän ja näyttölaitteen ohjelmiston toteutukselle. Käyttöliittymä sisältää mm. eleohjauksia LeapMotion-laitetta [18] hyödyntäen ja mahdollisesti rattiohjaimen sekä polkimet.

Budjetin osalta mitään erityisiä rajoitteita ei ollut, kunhan laitteiston hinta pysyy kohtuullisena.

2.2 Dokumentointi

Koska kyseessä on prototyyppi, jota mahdollisesti käytetään tulevaisuudessa varsinaisen tuotteen perustana on tärkeää, että dokumentointi tehdään hyvin. Ohjelmiston käytöstä ja rajapinnoista tehdään käyttöohjeet ja dokumentaatio. Näiden lisäksi ohjelmiston rakenne ja toiminta on dokumentoitu ohjelmakoodissa sekä diplomityössä. Projektin edetessä ilmenevät ongelmat ja mahdolliset jatkokehitysideat kirjataan myös ylös. Tavoitteena dokumentoinnissa on, että prototyypin käyttö olisi mahdollisimman helppoa ja että tuotteen jatkokehitys voidaan perustaa prototyyppiin.

2.3 Rajapinnat

Osa tässä työssä vaadittavista rajapinnoista oli projektin alkuvaiheessa melko hyvin tiedossa. Näitä olivat mm. traktorin CAN-väylän kuuntelu, kommunikointi käyttöliittymän kanssa, sekä lasien ohjaukseen käytetty SPI-yhteys (engl. Serial Peripheral Interface) [32]. Traktorin CAN-väylä käyttää J1939-standardin [31] mukaista protokollaa ja väylältä on tarkoitus ainoastaan kuunnella diagnostiikkaan liittyviä tietoja ja erilaisten parametrien arvoja, kuten kierrosluku, nopeus, öljynpaine yms. Käyttöliittymän kanssa kommunikoinnin vaatimuksena on luotettava ja helposti kahden laitteen välisen kommunikaatioon kykenevä tietoliikenneväylä, johon valittiin lähes joka laitteesta löytyvä Ethernet-yhteys [14]. Lasien ohjauskortit käyttävät SPI-protokollaa, joka asettaa rajoitteita yhteyden pituudelle ja nopeudelle. Ohjelmiston sisäisillä rajapinnoilla ei ollut erityisiä vaatimuksia ja ne riippuvat pitkälti valitusta toteutuksesta.

2.4 Laitteisto

Laitteiston tulisi olla sellainen, että siinä on rajapinnoissa mainitut ulkoiset liitynnät. Tähän tarkoitukseen on olemassa huomattava määrä erilaisia yhden piirilevyn tietokoneita, joista löytyy kaikki tarvittava samassa paketissa.

Prototyypin toimintaympäristö tulee olemaan sisätiloissa, joten mitään erityisiä suojaustarpeita ympäristöltä tavallisen koteloinnin lisäksi ei ole. Jatkokehityksen kannalta laitteistoksi on kuitenkin hyvä valita sellainen, josta on saatavilla versio myös ankarampaan ympäristöön. Käyttöjännitteen osalta ei ole vaatimuksia, kunhan se on mahdollista muuntaa sopivaksi traktorin sähköjärjestelmän jännitteestä (12 tai 24 volttia). Lasien ohjauspiirien käyttöjännite on 12 volttia.

Laitteiston ulkoisille mitoille ei ole erityisiä rajoituksia, kunhan se mahtuu messuilla esiteltävän demolaitteen rakenteisiin. Tämä ei tuota ongelmia, sillä nykyiset yhden piirilevyn tietokoneet ovat kooltaan hyvin kompakteja. Liityntöjen osalta minimivaatimuksena rajapintojen perusteella on, että laitteesta löytyvät SPI, CAN ja Ethernet. Tämän lisäksi on hyvä olla jonkinlainen debug-liityntä, jolla laitetta voidaan tarvittaessa käskyttää. Usein tähän tarkoitukseen käytetään RS-232-standardin [1] mukaista porttia.

Muita laitteiston valintaan vaikuttavia tekijöitä olivat valmistajan tuotetuki, mukana toimitettava ohjelmisto sekä dokumentaatio. Käytännössä kaikki laitevaihtoehdot ovat Linux-pohjaisia, joten mukana toimitettavaan ohjelmistoon tulee kuulua riit-

tävän uusi versio Linux-ytimestä, sekä tarvittavat ajurit SPI-, CAN-, ja Ethernet-oheislaitteille.

2.5 Ohjelmisto

Toteutettavan ohjelmiston suunnittelu perustuu aiemmin mainittuihin rajapintoihin. Tärkeimmät vaatimukset ohjelmistolle ovat, että vasteaika käyttöliittymältä tulevan komennon ja näytöllä näkyvän tiedon välillä on oltava käyttäjälle huomaamaton, CAN-väylän kuuntelun on oltava riittävän nopeaa, konfiguroitavuuden tulee olla helppoa ja ohjelmiston rakenteen tulee olla jatkokehitystä ajatellen geneerinen. Jotta vasteaika tarjoaisi mahdollisimman sulavan käyttäjäkokemuksen, on sen oltava alle 100 millisekuntia [6].

J1939-standardin mukaisessa CAN-väylässä tiedonsiirtonopeus on 250 kbit/s, jolloin paketteja voi tulla enintään noin 1900 sekunnissa olettaen, että paketin koko on yleensä 128 bittiä [17]. CAN-kuuntelijan on siis kyettävä lukemaan viestejä vähintään tällä nopeudella, mikäli kaikki viestit halutaan vastaanottaa. Joissakin tapauksissa näin ei välttämättä ole, sillä esimerkiksi yhden kierroslukutiedon jääminen välistä ei haittaa, mutta diagnostiikkaviestin katoamisella voi olla vakavampiakin seurauksia. Lisäksi ohjelmiston konfiguroinnin on tapahduttava siten, että esimerkiksi CAN-väylältä kuunneltavien viestien vaihtuessa ohjelmistoa ei tarvitse kääntää uudelleen, joten erillisen konfigurointiedoston käyttö on tarpeen.

Koska toteutuksessa tarvitaan paljon laitteistorajapinnan toimintoja, soveltuu ohjelmointikieleksi Linux-ympäristössä yleisesti käytetty C-kieli. Myös suorituskykyisyyden tarve tukee C-kielen valintaa, sillä vaikka yhden piirilevyn tietokoneet ovat nykyisin melko tehokkaita, ei ylemmän abstraktiotason enemmän resursseja kuluttavalle kielelle ole käyttöperusteita.

Ohjelmisto käännetään ristikäännösympäristössä, jonka jälkeen se siirretään ajettavaksi varsinaiseen laitteeseen. Ristikäännöstyökalujen lisäksi ohjelmointiympäristöstä on löydyttävä debug-työkalut ja versionhallinta.

3. KÄYTÖSSÄ OLEVA YMPÄRISTÖ JA TEKNIIKKA

Tässä luvussa esitellään työn toteutukseen valittu ympäristö ja tekniikka. Käytössä olevien tekniikoiden toimintaperiaatteet ja työn kannalta oleelliset ominaisuudet esitellään yleisellä tasolla. Tekniikoiden lisäksi esitellään käytettäväksi valittu laitteisto ja ohjelmiston kehitysympäristö.

3.1 CAN

CAN on vuonna 1986 SAE-kongressissa (engl. Society of Automotive Engineers) esitelty standardi [7], joka on saavuttanut suuren suosion autoteollisuudessa, ja sitä hyödynnetään myös muissakin ajoneuvoissa, kuten työkoneissa ja laivoissa. Myös teollisuusautomaation ohjauksessa CAN on löytänyt paikkansa johtuen sen yksinkertaisesta rakenteesta ja hyvästä virheensietokyvystä. CAN-standardin kehitys on valtaosin ollut Boschin ansiota, joka on julkaissut useita standardeja vuosien varrella. Näistä tunnetuin on Bosch CAN 2.0, johon nykyisin ISO-organisaation(engl. International Organization for Standardization) hallinnoima CAN-standardi pitkälti perustuu. CAN-standardi ISO-11898 määrittelee seitsemänkerroksisen OSI-mallin(engl. Open Systems Interconnection model) mukaisen protokollapinon kaksi alinta kerrosta, eli fyysisen kerroksen ja siirtokerroksen. Ylempien kerroksien toteutukseen on olemassa monia CAN-protokollan päälle rakennettuja protokollia, joihin myös tässä työssä käytetty J1939 kuuluu. CAN on protokollana yhteydetön ja kaikki viestit lähetetään yleislähetyksenä, joita kuka tahansa väylää kuunteleva voi vastaanottaa. Tämä mahdollistaa sen, että kaikki väylään liitetyt laitteet saavat viestit samanaikaisesti, mikä on tärkeää reaaliaikaisissa järjestelmissä.

3.1.1 Tiedonsiirto

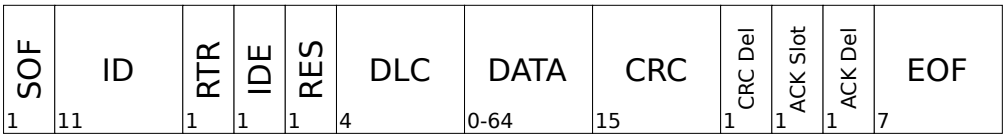
CAN-protokolla on moni-isäntäprotokolla, jossa väylällä on useita itsenäisesti toimivia laitteita samanaikaisesti ja väylän varaus perustuu CSMA-tekniikkaan (engl.

Carrier Sense Multiple Access) [19]. Samanaikaisesti lähettävien laitteiden poissulkeminen on toteutettu käyttämällä dominoivaa (looginen 0) ja resessiivistä (looginen 1) bittiä. Ennen lähetystä laite kuuntelee, onko väylä vapaa, jonka jälkeen se aloittaa lähetyksen todetessaan väylän vapaaksi. Lähettäessään laite kuuntelee myös väylää jolloin, jos resessiivistä bittiä lähettävä laite havaitsee väylän arvon dominoivaksi, se luopuu lähetyksestä ja yrittää myöhemmin uudelleen. Luopuessaan lähetyksestä se siirtyy vastaanottamaan linjalla olevaa lähetystä. Tällä tavalla voidaan myös priorisoida viestejä käyttämällä CAN-viestin alkupään bittejä osoitteistukseen ja priorisointiin. Tiedonsiirrossa CAN-väylällä käytetään NRZ-koodausta sekä täytebittejä, jossa viiden perättäisen samanarvoisen bitin jälkeen lisätään yksi bitti vastakkaisella arvolla. Näin varmistetaan riittävä määrä signaalitason muutoksia, joita käytetään laitteiden sisäisten kellojen synkronointiin.

Erilaisia viestityyppejä on CAN-standardissa määritelty neljä: data-, request-, error- ja overload-kehys. Näistä data-kehys on yleisimmin käytetty, sillä ainoastaan sillä voidaan kuljettaa tietoa. Request-kehystä käytetään silloin, jos haluttu data ei ole säännöllisin väliajoin automaattisesti lähetettävää dataa, vaan kysyttäessä lähetettävää. Error-kehys nimensä mukaisesti ilmaisee, mikäli väylällä on tapahtunut virhe. Overload-kehyksen lähettää sellainen laite, joka ei pysty käsittelemään kaikkia sille tarkoitettuja paketteja tai se havaitsee CAN-kehysten välisessä aikaikkunassa väylällä laittoman dominantin tilan.

Data- ja request-kehyksen rakenne on 11-bittisellä tunnisteella kuvan 3.1 mukainen. Kehys alkaa SOF-bitillä, joka on looginen 0. Tämän jälkeen tulee tunnistekentän 11 ensimmäistä bittiä. Koska looginen 0 on dominoiva, arvoltaan pienempi tunniste saa suuremman prioriteetin. Request-kehystä lähetettäessä RTR-kenttä on resessiivinen ja data-kehystä lähetettäessä dominoiva. IDE-kenttä ilmaisee resessiivisenä, että käytössä on 29-bittinen tunniste 11-bitin sijaan. RES-kenttä on varattu tulevaisuutta varten. DLC-kenttä ilmaisee DATA-kentän sisältämän datan määrän tavuina. Tarkistussumma sijaitsee dataosion jälkeen CRC-kentässä, jonka jälkeen vastaanottava laite kuittaa viestin vastaanotetuksi asettamalla väylän dominoivaksi ACK-kentän aikana. Lopuksi viesti päätetään EOF-kentällä, joka sisältää 7 resessiivistä bittiä. Ennen seuraavan viestin alkua tulee väylän olla vähintään 3 bitin ajan resessiivisessä tilassa.

Tunnisteen osalta kehysrakenteita on kahdenlaisia: 11-bittisellä tunnisteella ja 29-bittisellä tunnisteella. Kuvassa 3.2 on esitetty CAN-kehys 29-bittisellä tunnisteella. Alun perin 11-bittinen tunnistekenttä osoittautui liian lyhyeksi erilaisten viestien määrän kasvaessa, joten kehitettiin mahdollisuus käyttää 29-bittistä tunnistekenttää. Käytettäessä 29-bittistä kehystä asetetaan SRR- ja IDE-kenttä resessiiviseksi,



Kuva 3.1 *CAN-kehys, jonka tunnistekenttä on 11-bittinen. Kentän koko vasemmassa alareunassa bitteinä.*

Kuva 3.2 *CAN-kehys, jonka tunnistekenttä on 29-bittinen. Kentän koko vasemmassa alareunassa bitteinä.*

josta seuraa myös se, että viestit, joilla on 11-bittinen tunniste ovat prioriteetiltaan suurempia kuin 29-bittisellä tunnisteella olevat viestit. Kentät r1 ja r0 on varattu tulevaisuutta varten. 29-bittisillä ja 11-bittisillä tunnisteilla olevia viestejä voidaan lähettää samassa CAN-väylässä, kunhan 11-bittisiä viestejä tulkitsevat laitteet osaavat jättää huomiotta 29-bittisellä tunnisteella olevat viestit. [7]

3.1.2 Fyysinen rakenne

CAN-protokolla ei määrittele, miten väylä rakennetaan fyysisesti, ainoastaan sen, että väylällä on voitava esittää dominoiva ja resessiivinen bitti. Yleisin käytössä oleva fyysisen kerroksen standardi on ISO-11898-2, jossa CAN-väylä toteutetaan kahdella johtimella. Tiedonsiirtoon käytetyistä johtimista käytetään nimitystä CAN-High ja CAN-Low, joiden nimitys juontaa siitä, että CAN-High-johtimessa on suurempi potentiaali kuin CAN-Low-johtimessa. Viestien välitys perustuu näiden johtimien väliseen jännite-eroon. Resessiivisessä tilassa CAN-High-johdin on 3.5 volttia ja CAN-Low-johdin 1.5 volttia, jolloin jännite-ero on 2 volttia. Lähetettäessä dominoivaa bittiä CAN-High- ja CAN-Low-johtimet asetetaan kummatkin arvoon 2 volttia, jolloin jännite-ero on 0 volttia. Standardin mukaisesti väylään kuuluvat myös 120 ohmin päätevastukset, joiden tehtävänä on vähentää tiedonsiirrossa syntyviä heijastumia.

Topologialtaan väylän tulisi olla mahdollisimman lähellä yhtä johdinta, johon kaikki laitteet kiinnittyvät. Tämä ei kuitenkaan ole aina mahdollista esimerkiksi suurissa teollisuushalleissa, jolloin voidaan käyttää toistimia ja saada aikaan esimerkiksi tähtimäinen topologia. Tällöin pitää kuitenkin ottaa huomioon viestin etenemisviive, joka rajoittaa tiedonsiirtonopeutta. Ajoneuvoissa saatetaan käyttää useampaa CAN-väylää pitkistä etäisyyksistä tai turvallisuussyistä johtuen. Esimerkiksi kriit-

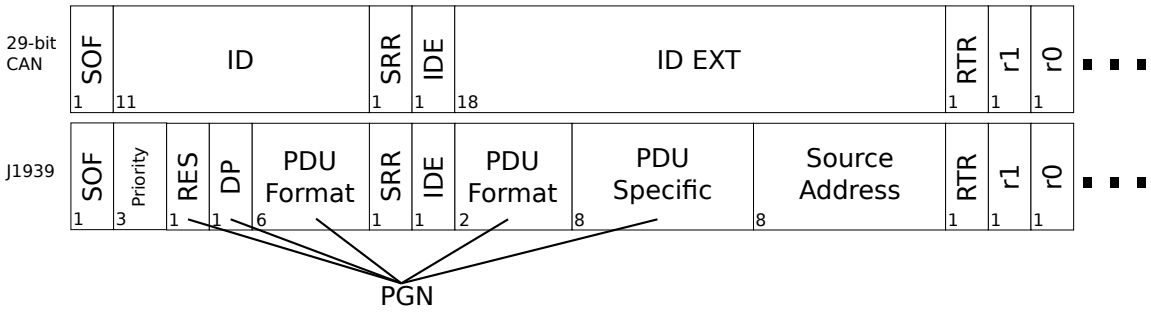
tistä tietoa välittävät moottorinohjausyksiköt voivat olla omassa CAN-väylässään ja vähemmän kriittiset laitteet omassa väylässään.

3.2 J1939

J1939 [31] on yksi monista CAN-protokollan päälle kehitetyistä protokollista, jotka lisäävät ominaisuuksia, joita ei CAN-protokollasta löydy. SAE kehitti sen yhtenäiseksi standardiksi ajoneuvossa olevien laitteiden kommunikointia varten, ja se määrittelee tarkoin minkälaista tietoa väylällä kuljetetaan ja missä muodossa. Esimerkiksi tieto moottorin kierrosluvusta löytyy aina samalla tunnisteella varustetun paketin yksilöidystä kohdasta. Näiden lisäksi se tarjoaa mm. osoitteistuksen, viestien lähettämisen haluttuun osoitteeseen ja suurempien kuin CAN-protokollan salliman 8 tavun pakettien siirron. Pääsääntöisesti J1939-protokollaa käytetään raskaammissa ajoneuvoissa ja työkoneissa, kuten traktoreissa ja muissa maatalouskoneissa. J1939 on toiminut myös pohjana muille protokollille, kuten merenkulussa käytetylle NMEA 2000 -protokollalle [21] ja sotilasajoneuvoissa käytetylle MilCAN-protokollalle [22].

3.2.1 Kehysrakenne

Kehysrakenne J1939-protokollassa käyttää CAN-protokollan extended-versiota, jossa tunnisteelle on varattu 29 bittiä tavallisen 11 bitin sijaan. Kuvassa 3.3 on esitetty sen kehysrakenne. Olennaista J1939-protokollassa on, että kaikki väylällä välitettävät parametrit, joita kutsutaan nimellä SPN (engl. Suspect Parameter Number) on jaettu parametriryhmiin PGN (eng. Parameter Group Number). PGN löytyy jokaisen paketin tunnisteesta, joka on jaettu kuuteen osaan. Ensimmäisenä on prioriteetti, joka on arvoltaan välillä 0–7 pienemmän arvon saadessa suuremman prioriteetin. Seuraavana on tulevaisuutta varten varattu reserved-bitti, sekä data page, jolla saadaan tarvittaessa laajennettua parametriryhmien määrää. Seuraavat kahdeksan ovat PDU format (engl. Parameter Data Unit), joka määrittelee onko kyseessä osoitettu viesti (arvot 0–239) vai yleislähetys (arvot 240–255). Seuraavan PDU Specific -osan kahdeksan bittiä toimivat PDU Format -osan perusteella joko kohdeosoitteena tai parametriryhmän laajennoksena (engl. Group extension). Viimeiset kahdeksan bittiä määrittelevät viestin lähdeosoitteen. PGN muodostuu reserved-bitistä, data page-bitistä sekä PDU Format- ja PDU Specific -osasta. Viestin dataosiosta löytyvät parametriryhmään kuuluvien parametrien arvot. Parametriryhmien parametrit ja niiden sijainti dataosiossa on määritelty J1939-standardin dokumentissa J1939/71 Vehicle Application Layer [29]. Joitakin PGN-arvoja on myös varattu valmistajien omaan käyttöön. [40]



Kuva 3.3 *J1939-kehyksen alkuosa ja miten se vertautuu CAN-kehykseen.*

3.2.2 Viestit ja osoitteistus

J1939-viestit voivat olla kaikille välitettäviä yleislähetyksiä tai jollekin laitteelle kohdistettuja viestejä. Jokaisella laitteella tulee olla oma osoite ja sen pitää rekisteröidä se omaan käyttöönsä Address Claim -viestillä. 8-bittisestä osoiteavaruudesta osoite 255 on varattu yleislähetyksille ja 254 sellaisille laitteille, joilla ei ole vielä omaa osoitetta, joten samanaikaisesti väylällä voi olla 254 laitetta. Yleensä jokaiselle laitteelle on ennalta määritelty jokin kiinteä osoite, mutta J1939 mahdollistaa myös dynaamisen osoitteistuksen. Jokainen J1939-paketti sisältää aina lähettäjän osoitteen.

Oman osoitteen lisäksi jokaisella laitteella on 8 tavun mittainen yksilöllinen nimi. Nimi on yhdistelmä erilaisia kenttiä, joihin kuuluvat mm. valmistajan tunniste, laitteen tarjoama toiminto ja laitteen yksilöllinen tunniste. Nimeä käytetään erityisesti dynaamisessa osoitteistuksessa, jolloin pienemmän nimen arvon omaamalla laitteella on suurempi prioriteetti osoitteita haettaessa.

Sellaisten viestien välittämiseen, jotka eivät mahdu tavallisen CAN-viestin kahdeksan tavun mittaiseen dataosioon käytetään J1939-protokollan tiedonsiirtoprotokollaa [28]. Käytännössä tämä on oma PGN, jonka dataosiossa useamman paketin mittaiset viestit välitetään. Useimmat standardissa määritellyt parametriryhmät mahtuvat yhteen kahdeksan tavun pakettiin, mutta esimerkiksi diagnostiikkaviestit tai valmistajakohtaiset viestit voivat ylittää kahdeksan tavun rajan.

J1939 tarjoaa paljon tietoa diagnostiikkaan liittyen [30], mutta tämän työn kannalta riittää, kun väylällä yleislähetyksenä lähetettävät aktiiviset diagnostiikkakoodit huomioidaan. Aktiiviset diagnostiikkakoodit sijaitsevat omassa parametriryhmässään DM1(engl. Diagnostic Message 1), jonka dataosiossa on lueteltu ne parametrinumerot, joissa vika on havaittu. Jos parametrinumeroita on niin monta, etteivät ne mahdu yhteen pakettiin, tiedonsiirrossa käytetään edellisessä kappaleessa mainittua tiedonsiirtoprotokollaa.

3.2.3 Linux-tuki

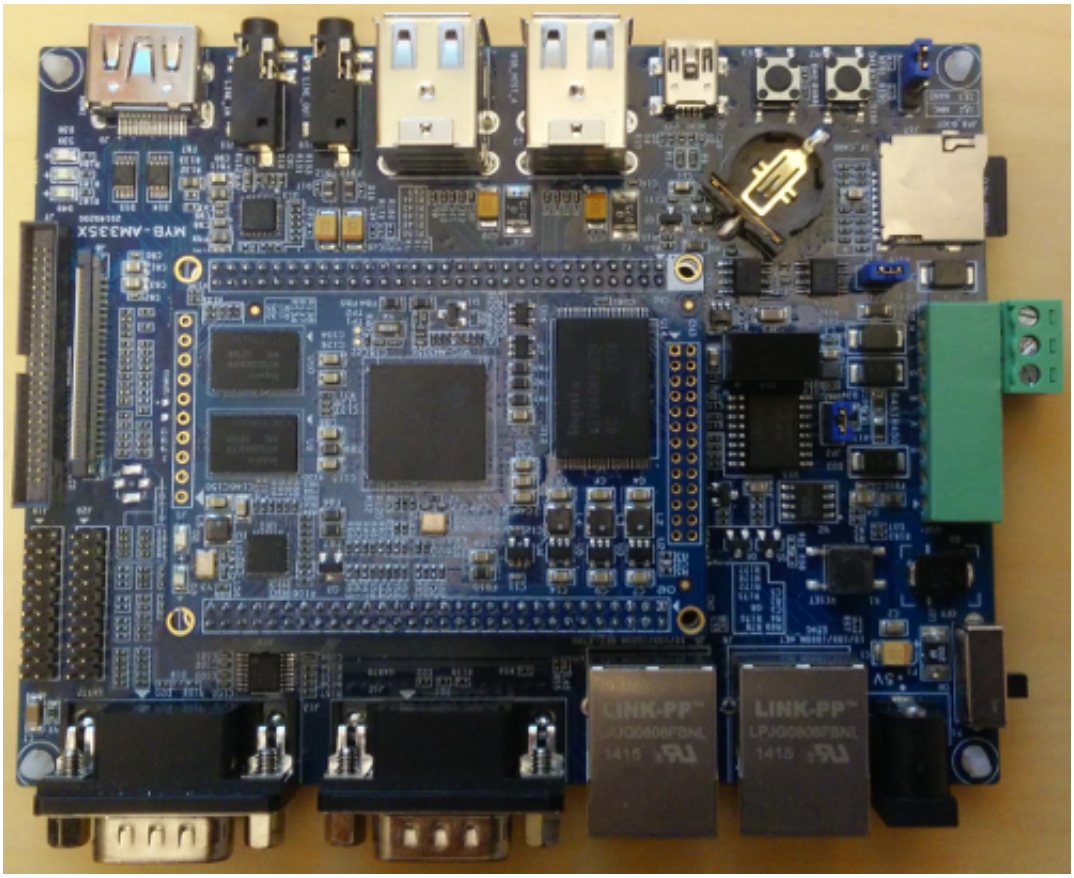
Linux-ytimeen on saatavilla Volkswagen Group -yhtiön kehittämänä CAN-protokollan tuki ja se löytyy jo valmiina useimmista uusista ytimen versioista. Tämän päälle EIA Electronics on kehittänyt tuen J1939-protokollalle [10]. Jotta J1939-tuen saa käyttöön, tulee tarvittavat moduulit kääntää ja asentaa, joka käytännössä tarkoittaa, että halutun ytimen version lähdekoodiin lisätään J1939-tukeen liittyvät osiot ja käännetään ydin uudelleen. J1939-protokollapino abstrahoi joitakin yksityiskohtia, kuten useamman paketin tiedonsiirtoon käytetyn protokollan, jolloin käyttäjälle näkyy ainoastaan paketin PGN, lähde- ja kohdeosoite sekä itse paketin sisältö.

3.3 Laitteisto

Projektin laitevalinnan osalta vaihtoehtoja oli runsaasti. Luvun 2.4 valintakriteerit täyttäviä yhden piirilevyn tietokoneita löytyy useita. Tästä johtuen painotettiin hyvää dokumentaatiota ja mahdollisuutta laitteen käyttöön jatkossa. Laitteistoksi valittiin lopulta kuvassa 3.4 oleva MYIR MYD-AM335X -kehitysalusta Texas Instruments AM3352 -prosessorilla [23]. Laitteesta löytyvät vaadittavat CAN-, Ethernet- ja SPI-yhteydet. Valintaa puolsi myös mahdollisuus ulkolämpötiloihin sopivaan laitteistoversioon. Laitteiston mukana toimitettiin Linux-ydin versiolla 3.2.0, johon lisättiin tuki J1939-protokollalle. Linux-lähdekoodin lisäksi laitteiston mukana toimitettiin kattava dokumentaatio ja työkalut ohjelmistojen kehitykseen ja niiden siirtoon laitteistolle.

Kokeilumielessä tilattiin myös Embest DevKit8600 -kehitysalusta [26], jota käytettiin ohjelmiston testauksessa ennen varsinaisen alustan saapumista. Myöhemmin sitä hyödynnettiin CAN-väylän testauksessa, jossa sen avulla lähetettiin viestejä toiselle kehitysalustalle. Verrattuna projektin varsinaiseen kehitysalustaan dokumentaatio oli tällä alustalla puutteellinen eikä kokonaisuus ollut yhtä viimeistelty. Linux-ytimen versio oli eri (3.1.0), joten J1939-tuen saamiseksi se piti kääntää erikseen tälle laitteelle.

Käytössä oli myös oskilloskooppi digitaali- ja analogiatuloilla. Se osoittautui erittäin hyödylliseksi testattaessa SPI-yhteyden toimivuutta ja tarkasteltaessa signaalin muotoa pitkillä johdinpituuksilla. Oskilloskoopilla voitiin myös mitata ajurin ajoituksia ja vertailla, miten erilaiset ohjelmalliset ratkaisut vaikuttivat nopeuteen.

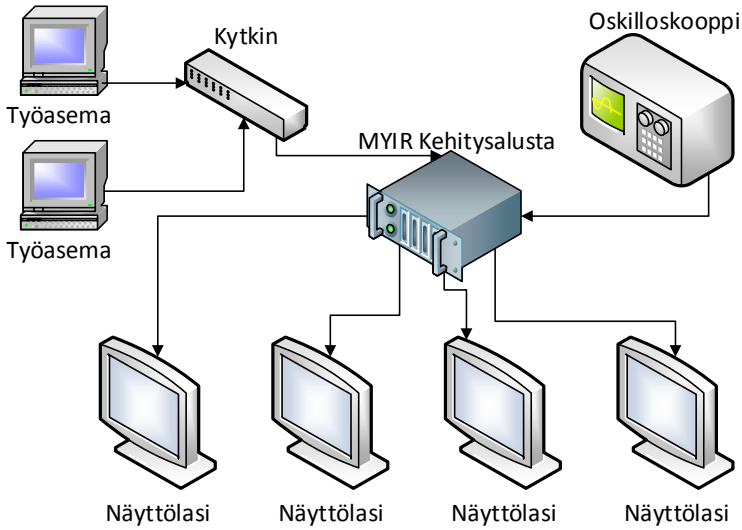


Kuva 3.4 *MYIR MYD-AM335X kehitysalusta.*

3.4 Kehitysympäristö

Kehitysympäristöä ei ennen projektin aloitusta oltu erityisemmin suunniteltu muuten kuin, että ohjelmiston kääntämiseen käytettäisiin ristikäännösympäristöä ja työkoneet olisivat Linux-pohjaisia. Projektin edetessä kuitenkin kehitysympäristöäkin kehitettiin. Aluksi oli tarkoitus, että käännöstyökalut asennetaan työkoneille. Tämä kuitenkin osoittautui melko hankalaksi käyttöoikeuksien rajoitteiden vuoksi. Ratkaisuna tähän ongelmaan käännöstyökalut asennettiin virtuaalikoneisiin, mikä myös teki ympäristön siirrettävyydestä huomattavasti helpompaa. Virtuaalikoneen käyttöä suositeltiin myös kehitysalustan toimittajan dokumentaatiossa. Ristikäännöstyökalut toimitettiin kehitysalustan mukana ja virtuaalikoneen käyttöjärjestelmänä käytimme Ubuntun versiota 14.04.

Ohjelmointiympäristönä toimi aluksi tavallinen tekstieditori, mutta myöhemmin päädyimme käyttämään QtCreator-työkalua [38]. Syy miksi päädyimme QtCreator-työkaluun oli se, että ohjelmakoodin muokkaaminen ja tiedon etsiminen oli helpompaa sellaisen työkalun avulla, joka osaa esimerkiksi siirtyä funktion toteutukseen nimen perusteella kutsukohdasta. Tähän tarkoitukseen olisi myös käynyt mikä tahan-



Kuva 3.5 *Kehitysympäristön rakenne.*

sa IDE-ohjelmisto (engl. Integrated Development Environment). Lisäksi käännöstyötä pyrittiin automatisoimaan mahdollisimman paljon skriptien avulla. Virheenkorjausta ja testausta varten kehitysalustalle käännettiin myös GDB (engl. The GNU Project Debugger), jonka avulla voitiin tarkastella ohjelman ajoaikaista toimintaa. Kaiken ohjelmiston ja dokumentaation ylläpitoon käytettiin SVN-versionhallintaa.

Alkuvaiheessa käännetyt ohjelmat siirrettiin USB-muistitikulla kehitysalustalle ajettavaksi ja ohjelman tulosteita tarkkailtiin sarjaporttiyhteyden avulla. Tämä ei ollut mitenkään tehokas ratkaisu varsinkaan, kun kehittäjiä oli kaksi. Päädyimme rakentamaan sisäverkon, jossa työkoneet ja kehitysalusta ovat samassa verkossa ja tiedonsiirto tapahtuu SSH-yhteyden (engl. Secure Shell) yli. Näin käännetyt tiedostot saatiin helposti siirrettyä kohdelaitteelle ja ohjelmiston testaus ja debuggaus voitiin suorittaa verkon yli. Kuvassa 3.5 näkyy koko kehitysympäristön rakenne.

3.5 YAML-konfiguraatio

Järjestelmän helppo muunneltavuus eri näytöille ja CAN-viesteille oli yksi vaatimuksista johtuen osittain siitä, että projektin alkuvaiheessa ei ollut vielä tiedossa tarkalleen mitä näytöillä on tarkoitus näyttää. Tulevaisuutta ajatellen on hyvä, että järjestelmä voidaan konfiguroida toimimaan erilaisilla näytöillä ja näyttämään eri viestejä. Tähän tarkoitukseen valitsimme YAML-merkintäkielen (engl. YAML Ain't Markup Language) [2].

YAML on tarkoitettu ihmisystävälliseksi merkintäkieleksi. Se on saanut vaikutteita useammasta eri kielestä, kuten XML, JSON, Perl, C jne. YAML:n tavoitteena on pyrkiä mallintamaan usein ohjelmointikielissä käytettyjä tietorakenteita, kuten listoja, taulukoita ja skalaareja siten, että ne ovat ihmiselle helppolukuisessa muodossa. Käytännössä tämä tarkoittaa, että erikoismerkkien määrä on pyritty minimoimaan ja tietorakenteiden rakenne ilmaistaan sisennyksillä. Esimerkki YAML:sta on listauksessa 3.1, jossa on luotu lista, jolla on alkioita. Esimerkissä listalla *lista* on kolme alkiota: *alkio1*, *alkio2* ja *alkio3*. Alkioilla *alkio1* ja *alkio2* on arvot *arvo1* ja *arvo2*. Alkiolla *alkio2* on lisäksi toinen alialkio, jonka sisältönä on arvot *arvo3* ja *arvo4* sisältävä lista. Kolmannen alkion *alkio3* sisältö on myös lista.

```
skalaari: arvo
lista:
        - alkio1: arvo1
        - alkio2: arvo2
          alialkio: [arvo3, arvo4]
        - alkio3: [arvo5, arvo6]
```

Ohjelma 3.1 *Esimerkki YAML:sta.*

YAML:n käyttöön on olemassa useita kirjastoja eri ohjelmointikielille [12]. Tähän projektiin valitsimme C-kielellä toteutetun libyaml-kirjaston [27]. Se tarjoaa rajapinnan, jolla YAML-tiedosto voidaan lukea alusta loppuun sekventiaalisesti samalla erotellen YAML:n rakenteen nimet ja arvot. Libyaml ilmaisee tiedostossa etenemisen erilaisina tapahtumina, joista esimerkiksi listan alku ja loppu ovat omat tapahtumansa ja jokaisesta niiden välillä esiintyvällä alkiolla ja arvolla on oma tapahtumansa.

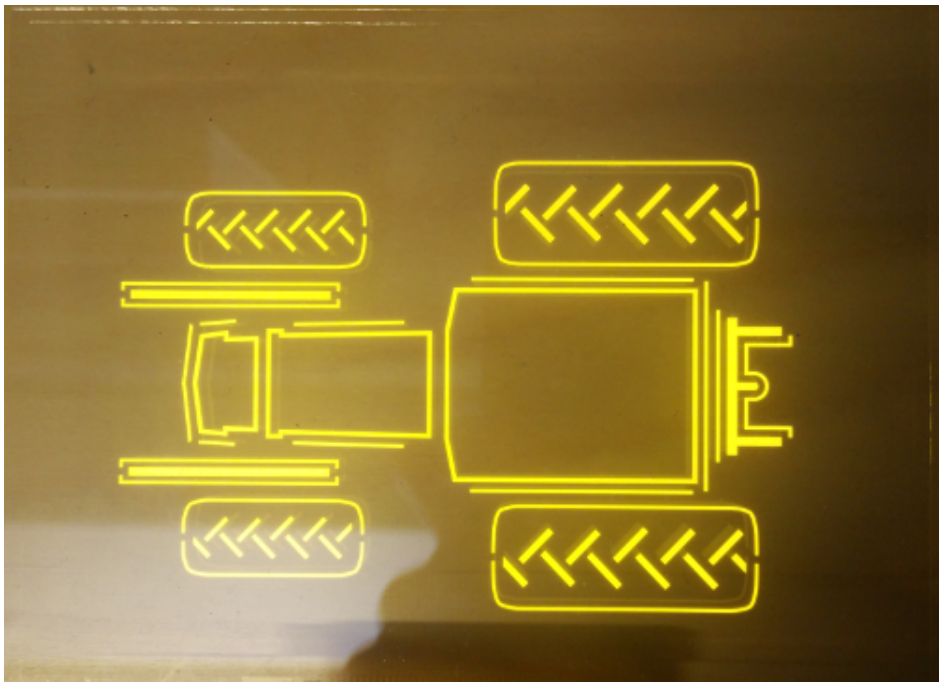
3.6 Näyttölasi

Näyttölasit ovat Beneq Products Oy:n valmistamia Lumineq TASEL-näyttöjä (engl. Transparent Electroluminescent Display) [3], jotka ovat pois päältä ollessaan läpinäkyviä. Ne sisältävät väriltään keltaisia kiinteitä segmenttejä, joita voidaan sytyttää ja sammuttaa. Näyttölaseja tässä projektissa on neljä ja jokaisella on oma ohjauskorttinsa, jota ohjataan SPI-väylän kautta. Kuvassa 3.6 näkyvässä ensimmäisessä näytössä on kommunikointiin liittyviä ikoneja, kuten puhelin, gps ja wlan. Sillä on tarkoitus ilmaista kuljettajalle esimerkiksi puhelimeen saapuneesta viestistä. Kuvassa 3.7 näkyvässä toisessa näytössä on kuva traktorista ja sitä on tarkoitus käyttää näyttämään diagnostiikkatietoja silloin, kun CAN-väylältä saadaan viesti viasta. Kuvan 3.8 kolmannessa näytössä on etäisyyteen liittyvä lukunäyttö, sekä nuolet

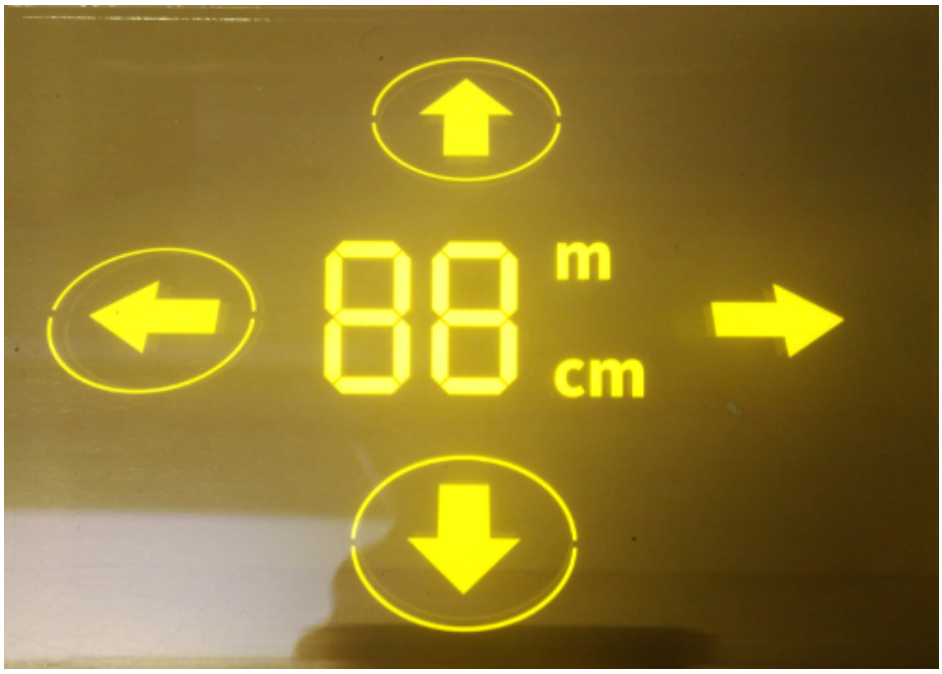
Kuva 3.6 *Ensimmäinen näyttölasi, joka sisältää kommunikointiin liittyviä ikoneja.*

eri suuntiin, joita voidaan käyttää ilmaisemaan esimerkiksi etäisyys johonkin kohteeseen. Kuvassa 3.9 olevassa neljännessä näytössä on nelinumeroinen lukunäyttö sekä ikoneita, jotka liittyvät traktorin nykyiseen tilaan. Tällä näytöllä on tarkoitus näyttää CAN-väylältä saatua traktorin tilatietoa, joita ovat esimerkiksi kierrosluku, nopeus ja jäähdytysnesteen lämpötila.

Näyttöjen ohjaukseen käytetyssä SPI-väylässä on kaikille väylään liitetyille laitteille yhteinen kellosignaali ja dataväylä, jossa tieto välitetään sarjamuodossa. Jokaisella väylään liitetyllä laitteella on oma erillinen valintasignaalinsa, jonka aktivoimisella lähetettävä viesti kohdistetaan oikealle laitteelle. Koska valitulla kehitysalustalla on ulostulot vain yhdelle kellosignaalille ja yhdelle datasignaalille, täytyy ne jakaa näytöille erillisellä jakajalla. Jakaja suunniteltiin siten, että se välittää näytöille myös niiden tarvitseman 12 V:n käyttöjännitteen, jolloin jakajalta lähtee jokaiselle näytölle yksi kaapeli.



Kuva 3.7 *Toinen näyttölasi, traktorin kuvan diagnostiikkatarkoituksiin.*



Kuva 3.8 *Kolmas näyttölasi, jolla voidaan ilmaista esimerkiksi etäisyyksiä johonkin kohteeseen.*

Kuva 3.9 *Neljäs näyttölasi, joka sisältää traktorin tilaan liittyvää tietoa.*

4. RATKAISUVAIHTOEHDOT

Tässä luvussa esitellään mahdollisia ratkaisuja työn toteutukseen ohjelmiston kannalta ja lopuksi kerrotaan mihin ratkaisuun päädyttiin ja mitkä ovat ratkaisuun johtaneet perustelut. Ohjelmiston rajapinnoista tiedetään, että käyttöliittymän ohjelmisto ajetaan eri laitteistolla ja käskyt tulevat TCP-yhteyden kautta. Näyttöjen rajapintana on SPI-väylä ja kolmantena rajapintana on traktorin CAN-väylä. Nämä yhdistävän ohjelmiston rakenne ja toiminta ovat päätettävissä.

Alustavien suunnitelmien mukaan ohjelmiston on tarkoitus vastaanottaa yksinkertaisia käskyjä käyttöliittymältä, joilla voidaan siirtyä eri tilojen välillä. Näytöt sisältävät ikoneita, joten luonnollisesti tilasiirtymät voidaan hoitaa valitsemalla eri ikoneita. Työnjako on toteutettu siten, että tässä diplomityössä toteutetaan TCP-yhteyttä ja CAN-väylää kuuntelevat osat ja se, miten toimitaan saapuneiden viestien perusteella. Toisessa diplomityössä toteutetaan näyttöjen ajuri ja siihen liittyvät toiminnot, kuten ajurin rajapinnan abstrahointi siten, että käyttäjän ei tarvitse tietää yksittäisten segmenttien numeroita, vaan segmentit voidaan nimetä ja tarvittaessa muodostaa niistä ryhmiä. Segmenttiryhmiä voidaan ohjata kuten yksittäisiä segmenttejä. Myös numeroiden näyttäminen 7-segmenttinäytöillä on toteutettu toisessa diplomityössä.

4.1 Ajuri

Jotta näyttöjä voitaisiin ohjata SPI-väylällä tarvitaan niille jonkinlainen ajuri. Se voi olla niin sanottu käyttäjätilan ajuri, jossa ohjaus toteutetaan ohjelmistolla, joka hyödyntää valmiina olevia Linuxin SPI-toimintoja ja ajureita tai oikea etuoikeutetussa tilassa toimiva ajuri. Ensimmäinen vaihtoehto voitaisiin integroida jopa suoraan itse ohjelmistoon ja se olisi jossain määrin helpompi toteuttaa, mutta myöhemmin esitetyt rajoitteet estävät tämän toteutustavan. Jälkimmäisessä vaihtoehdossa toteutus on vaikeampi ja virheet ajurissa vaikuttavat koko järjestelmän toimintaan, joten testaus on tärkeää. Testaus on myös hankalampaa, sillä etuoikeutetussa tilassa ajettavan ohjelmiston debuggaustyökalut ovat rajalliset (käytännössä pelkkä tulostusfunktio, ellei koko ydintä käännetä uudelleen debug-ominaisuuksilla).

Kehitysalusta asettaa joitakin rajoitteita näyttöjen ohjaukselle. Ensimmäisenä mainittakoon, että sisäänrakennettu SPI-oheislaite sisältää vain yhden valintasignaalin ja näyttöjä on neljä, joten tähän tarkoitukseen se ei sellaisenaan sovi. GPIO-portteja (engl. General Purpose Input/Output) sen sijaan löytyy runsaasti, ja niitä voidaan käyttää valintasignaalin tapaan. Siirrettäessä tietoa valintasignaali asetetaan loogiseen arvoon 0 ja lopetettaessa se palautetaan loogiseen arvoon 1. Kehitysalustan mukana toimitettu Linux-versio ja SPI-oheislaite eivät suoraan tue GPIO-porttien käyttöä valintasignaaleina, joten se pitää tehdä erikseen. Tämä on syy siihen, että ajuri pitää toteuttaa etuoikeutetussa tilassa, sillä tarjolla olevat valmiit yleisajurit, kuten spidev [37], eivät tue GPIO-porttien käyttöä. Myös käytetyn Linux-version oma rajapinta oli puutteellinen GPIO-porttien suhteen, joten niitäkin päädyttiin käyttämään suoraan muokkaamalla prosessorin rekistereitä.

Prosessorin rekistereiden suora käyttö vaikeuttaa ajurin uudelleenkäyttöä toisenlaisessa laitteessa, mutta ei kuitenkaan ole kovin suuri työ muokata ohjelmakoodia kohdelaitteelle sopivaksi. Ytimen versiosta 3.2.6 lähtien on myös Linuxin SPI-rajapinnassa suora tuki GPIO-porttien käyttöön valintasignaaleina. Etuoikeutetussa tilassa toimivan ajurin toteutusta tukee myös se, että se on nopeampi, koska etuoikeutetun tilan ohjelmistolla on suurempi prioriteetti ja se pääseee käsiksi suoraan prosessorin rekistereihin.

Ajurista ei kannata tehdä liian monimutkaista, vaan sen rajapinnan tulisi olla mahdollisimman yksinkertainen ja tarjota näyttöjen perusohjaus, kuten segmenttien sytytys ja sammutus. Ylemmän tason logiikka on helpompi toteuttaa erillisessä ajuria käyttävässä ohjelmassa ja sen muokkaaminen on helpompaa käyttäjätilassa.

4.2 Arkkitehtuuri

Ohjelmiston arkkitehtuurilla tässä yhteydessä tarkoitetaan sitä, miten ohjelmisto on jaettu osiin ja mitä sen eri osat tekevät. Edellisen kappaleen perusteella ajuri toteutetaan erillään muusta ohjelmistosta, joten päätettäväksi jää, millainen ajuria käyttävä ohjelmisto toteutetaan.

4.2.1 Yksi ohjelma

Rakenteeltaan yksinkertaisin vaihtoehto olisi toteuttaa koko ohjelmisto yhtenä prosessina, joka käsittelee käyttöliittymän viestit, CAN-väylän viestit ja ohjaa näyttöjä

ajurin avulla. Tällä tavalla toteutettuna ohjelmiston käyttäminen olisi yksinkertaista, sillä tarvitsisi vain käynnistää yksi ohjelma ja se hoitaisi kaiken. Ohjelmiston kehittäminen vaikeutuisi hieman verrattuna myöhemmin esiteltyihin toteutustapoihin testauksen, työnjaon ja jatkokehityksen kannalta. Testaus tulisi tehdä sellaisella ohjelmakoodilla, joka on ajettavissa, jolloin pienten muutosten nopea testaaminen hankaloituu. Tähän ratkaisuna voisi olla, että kummallakin kehittäjällä on oma versio, jolla voi testata pienet muutokset ja myöhemmin ne yhdistetään yhdeksi versioksi. Jatkokehityksen ja toiminnallisuuden lisäämisen kannalta tämä on huono vaihtoehto, sillä koko ohjelmaa pitää muokata ja pahimmassa tapauksessa se voidaan joutua tekemään kokonaan uudelleen.

4.2.2 Erilliset ohjelmat

Eriyttämällä ohjelmisto toimintojen perusteella eri prosesseiksi voidaan selkiyttää sen toimintaa ja tehdä siitä modulaarisempi. Käytettäessä erillisiä prosesseja voidaan niiden väliset rajapinnat sopia etukäteen ja lopuksi testata kokonaisuuden toiminta. Etukäteen sovitut rajapinnat mahdollistavat helpomman ohjelmiston jatkokehittämisen ja eri moduulien toiminnan muokkaamisen. Ohjelmiston kehittäminen on myös helpompaa, sillä ohjelmakoodeja ei tarvitse yhdistää ja kunkin osion testaaminen voidaan suorittaa erillään muista. Huonona puolena usean prosessin arkkitehtuurissa rajapintojen määrä kasvaa ja sen myötä prosessien väliseen kommunikaatioon kuluu enemmän resursseja. Ohjelmisto tulee kuitenkin olemaan toiminnallisuudeltaan melko yksinkertainen ja valittu kehitysalusta on suorituskyvyltään hyvä, joten tästä ei pitäisi muodostua ongelmaa.

Toiminnan perusteella ohjelmiston jako eri prosesseihin on tässä tapauksessa helppoa, sillä käsiteltävä data tulee useasta lähteestä. Tällöin prosessien ei tarvitse käsitellä samanaikaisesti samaa dataa, eikä rinnakkaisuuteen liittyviä ongelmia synny. Jako voitaisiin tehdä siten, että yksi prosessi käsittelee näyttöjen ohjausta ja toinen prosessi toimii kontrolliyksikkönä ottaen vastaan syötteitä CAN-väylältä sekä käyttöliittymältä ja lähettäen viestejä näyttöjä ohjaavalle ohjelmistolle. Toinen mahdollinen toteutustapa voisi olla, että näyttöä ohjaavan prosessin lisäksi CAN-väylälle ja käyttöliittymälle olisivat omat prosessit ja yksi kontrolliyksikkönä toimiva prosessi sisältäisi toimintalogiikan. Tähän voitaisiin myös käyttää kontrolliyksikön prosessissa säikeitä.

4.3 Yksi vai useampi säie

Usean säikeen käyttö voi tuoda joitakin etuja verrattuna yhden säikeen käyttöön. Verrattuna monisäikeiseen ohjelmistoon mitään sellaista toimintoa tai algoritmia ei kuitenkaan ole, jota yksisäikeisellä ohjelmalla ei pystyttäisi toteuttamaan. Church-Turingin konjektuurin [9] mukaan kaikki laskettava voidaan laskea yksinauhaisella Turing-koneella ja koska C-kieli on Turing-täydellinen, tästä seuraa, että yksisäikeisellä ohjelmalla voidaan toteuttaa samat asiat kuin monisäikeiselläkin. Monisäikeinen ohjelma voi kuitenkin käyttää tehokkaammin käytettävissä olevia resursseja. Erityisesti usean prosessorin järjestelmissä monisäikeistä ohjelmaa voidaan ajaa aidosti rinnakkain eri prosessoreilla. Myös yhden prosessorin järjestelmissä jotkin säikeet voivat jäädä odottamaan jonkin resurssin vapautumista tai viestin saapumista. Sillä välin muut säikeet voidaan pitää ajossa. Laitteistoksi valittu kehitysalusta sisältää yksiytimisen ARM-arkkitehtuurin prosessorin, joten aidosti rinnakkaista toteutusta ei voida saavuttaa.

Toteutuksen kannalta käytössä voisi olla useampi säie tai kaikki tehtäisiin yhdessä silmukassa. Yhden säikeen tapauksessa monet Linux-käyttöjärjestelmän tarjoamista funktioista ovat odottavia, joten ne pitäisi ajaa ei-odottavina ja huolehtia muulla tavoin siitä, ettei silmukassa pyörivä ohjelma kuluta liikaa resursseja. Tämä voidaan tehdä esimerkiksi nukuttamalla prosessi hetkeksi, jos mitään tehtävää ei ole, jolla varmistetaan, että käyttöjärjestelmän vuorontaja päästää myös muita prosesseja ajoon. Prosessorille jatkuvan silmukan pyörittämisen aiheuttama kuorma näkyy myös virrankulutuksessa, mikä on sulautetuissa järjestelmissä usein tärkeä kriteeri varsinkin akkukäyttöisissä laitteissa. Tässä tapauksessa kyse on kuitenkin ajoneuvosta, joka tuottaa tarvitsemansa sähkön polttomoottorin voimalla ja laitteiston tehonkulutus on melko pieni (n. 30 W näytöt mukaan lukien), joten virrankulutuksen minimointi ei ole avainasemassa.

4.4 Konfiguroitavuus

Tässä projektissa konfiguroitavuudella pyritään siihen, ettei ohjelmistoa tarvitsisi kääntää uudelleen esimerkiksi käytettäessä erilaisia näyttöjä. Käyttäjän kannalta ohjelman käyttökohteiden määrä kasvaa, mutta ohjelmoijan kannalta tämä merkitsee jonkin verran lisää työtä. Konfigurointi toteutetaan usein erillisellä tiedostolla, joka luetaan ohjelman alussa ja sen sisältö talletetaan ohjelman ajoaikaiseen muistiin.

Ohjelmiston konfiguroitavuuden lisääminen luonnollisesti kasvattaa erilaisten vaihtoehtojen huomioimisen tarvetta ohjelmistoa suunniteltaessa. Osan toiminnoista on

oltava melko geneerisiä, jotta useamman vaihtoehdon toteutus onnistuisi samalla koodilla ja tästä seurauksena koodin määrä kasvaa. Tosin suurin osa on yksinkertaisia ehtolausekkeita, joilla valitaan toiminto riippuen konfiguraatiosta. Myöskään kaikkea toiminnallisuutta ei kannata tehdä konfiguroitavaksi, sillä silloin konfiguraatiotiedoston rooli muuttuu lähemmäksi ohjelmointikieltä ja alkuperäinen ohjelmisto toimii tälle tulkkina, jolloin tavoite helposta konfiguroitavuudesta jää saavuttamatta.

Huomioitavaa on myös se, että konfiguroitavuuden kasvattaminen lisää myös mahdollisuuksia virheisiin. Ohjelmisto monimutkaistuu ja käyttäjän rajapinta kasvaa, jolloin esimerkiksi virheellisen konfiguraation mahdollisuus kasvaa. Virheensietokyky onkin yksi merkittävä kysymys ohjelmaa suunniteltaessa. Lopetetaanko ajo heti virheen ilmaantuessa, jatketaanko suorittaen korjaavia toimenpiteitä vai jatketaanko välittämättä virheestä? Se miten toimitaan, riippuu paljon virheen laadusta.

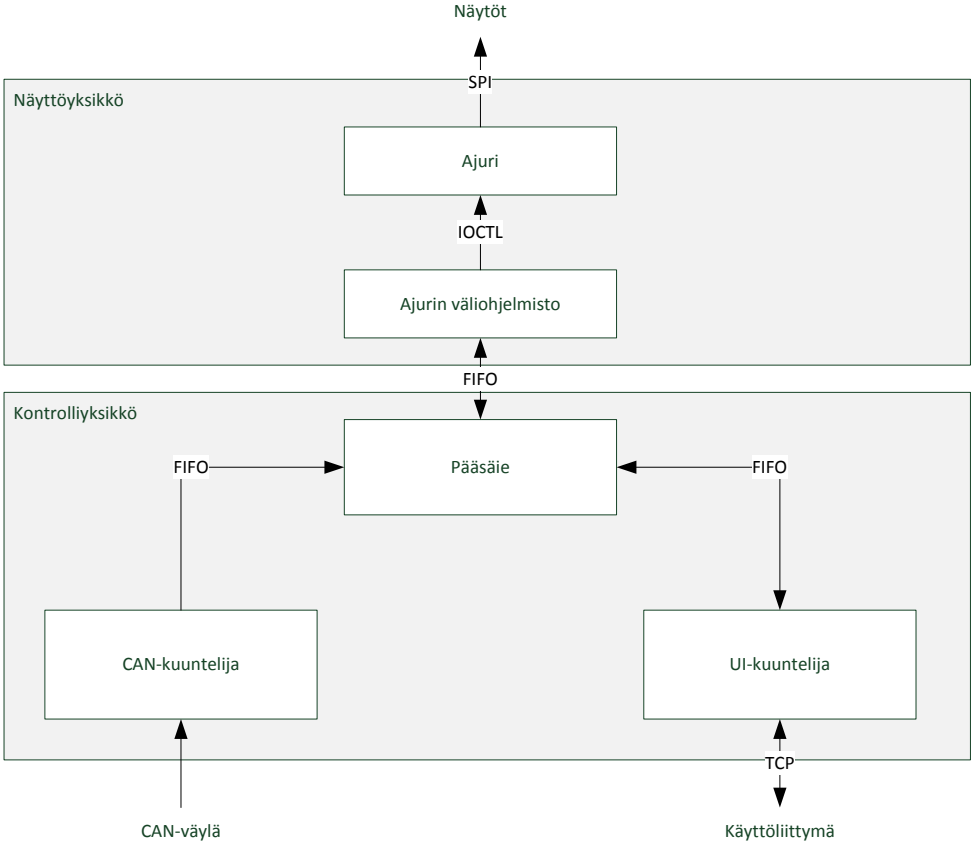
Vaikka konfiguroitavuudella on joitakin haittapuolia, sen tuomat hyödyt ovat suuret. Ohjelmiston muokattavuus eri tilanteisiin, tässä tapauksessa eri näytöille ja CAN-viesteille, on nopeampaa. Konfiguraatiotiedoston muokkaukset voidaan tehdä millä tahansa tekstieditorilla ja uuden konfiguraation voimaantulo edellyttää ainoastaan ohjelmiston uudelleenkäynnistyksen.

Toinen vaihtoehto konfiguraatiotiedostojen käytölle voisi olla käyttää käännösaikaista konfigurointia, jolloin ohjelmakoodi sisältäisi ehtoja, joita voidaan muokata käännösvaiheessa. Ohjelmiston käännösprosessi ja päivitys tulisi olla helposti tehtävissä. Tällaisessa ratkaisussa käyttäjällä ei olisi niin suurta rajapintaa ja käyttö olisi yksinkertaisempaa. Tosin muutosten tekeminen vaatisi lähes aina ohjelmiston uudelleenkääntämisen ja siihen ei käyttäjällä yleensä ole mahdollisuutta.

4.5 Valittu toteutus

Toteutuksen valinta tehtiin ohjelmiston vaatimusten ja jatkokehitysmahdollisuuksien pohjalta. Ohjelmiston arkkitehtuuri ja sen myötä tarvittavat rajapinnat suunniteltiin alustavasti yhdessä ennen toteutuksen aloitusta. Joiltakin osin rajapinnat ja toiminnallisuus muuttuivat tai niihin tehtiin lisäyksiä, mutta pääpiirteiltään ne vastasivat suunniteltua.

Toteutuksessa päädyttiin kuvan 4.1 mukaiseen arkkitehtuuriin. Ylemmässä osassa näkyy toisen diplomityön osuus ja alemmassa tämän diplomityön osuus. Ohjelmisto on jaettu siten, että ajuria käyttävä ohjelma on erillään CAN-väylää ja käyttöliittymää kuuntelevasta ohjelmasta. Näiden väliseen kaksisuuntaiseen kommunikointiin



Kuva 4.1 *Valitun toteutuksen arkkitehtuuri.*

käytettiin FIFO-jonoa (engl. First In First Out).

Tässä työssä toteutettu ohjelma on jaettu kolmeen säikeeseen, joista yksi kuuntelee CAN-väylää (CAN-kuuntelija), toinen käyttöliittymän TCP-yhteyttä (UI-kuuntelija) ja kolmas toimii tilakoneen tavoin kuunnellen CAN-kuuntelijan ja UI-kuuntelijan viestejä. Saatujen viestien perusteella se lähettää ajuria käyttävälle ohjelmalle käskyjä. Tarvittaessa esimerkiksi virheviestit välitetään myös käyttöliittymälle. Tämä toteutus mahdollistaa esimerkiksi useamman CAN-väylän kuuntelun tai useamman TCP-yhteyden kuuntelun luomalla vain oma säie jokaiselle kuunneltavalle tietoliikenneväylälle. Toteutus on tarkemmin kuvattu luvussa 5.

5. TOTEUTUS

Tässä luvussa esitellään tarkemmin edellisen luvun vaihtoehdoista valittu lopullinen toteutus. Ensimmäisenä esitetään tarkempi kuvaus ohjelmiston konfiguroinnista, jonka jälkeen kuvataan rajapinnat. Tämän jälkeen esitellään varsinaisen kontrolliyksikön rakenne ja toiminta. Viimeisenä tarkastellaan käytettyjä testausmenetelmiä ja dokumentaatiota.

5.1 Konfigurointi

Helppo konfiguroitavuus oli eräs tärkeimmistä vaatimuksista ohjelmistoa suunniteltaessa ja siihen tarkoitukseen valittiin käytettäväksi YAML-formaatissa olevat konfiguraatiotiedostot. Alun perin tiedostoja oli kolme: yksi CAN-kuuntelijalle, yksi käyttöliittymän kuuntelijalle ja yksi näytöille. Myöhemmin näyttöjen konfiguraatio jaettiin viiteen eri tiedostoon kokonaisuuden selkiyttämiseksi. Tarkempi kuvaus konfiguroinnista löytyy liitteestä 1.

5.1.1 Näyttöjen konfigurointi

Näyttöjen konfiguraatiossa jokaiselle segmentille on annettu nimi sekä tiedot, missä näytössä se sijaitsee ja missä kohdassa kyseistä näyttöä. Ohjelmisto lukee konfiguraatiotiedoston käynnistyksen yhteydessä ja jokaiselle segmentille luodaan tunniste, joka on juokseva numero konfiguraatiotiedoston järjestyksen perusteella. Tätä tunnistetta käytetään näyttölasin tulkin rajapinnassa ohjattavan segmentin tunnisteena. Tällä tavalla käyttäjälle näkyy vain lista kaikista segmenteistä, eikä tietoa niiden tarkemmasta sijainnista tarvita.

Useammasta segmentistä voidaan muodostaa ryhmä lisäten konfiguraatiotiedostoon kunkin ryhmän kohdalle lista siihen kuuluvien segmenttien nimistä. Ikonit ovat myös ryhmä useampia segmenttejä. Erona ryhmään on kuitenkin se, että ikoneita käytetään eri tilojen välillä navigointiin. Käytännössä jokainen ikoni on yksi kontrolliyksikön tila ja navigoinnissa siirrytään toiseen tilaan valitsemalla jokin toinen ikoni. Ikonin valinta ei aina vaikuta kontrolliyksikön tilaan, sillä niihin ei liity CAN-väylältä

kuunneltavia viestejä. Esimerkiksi Bluetooth-ikonin valinta ei vaikuta neljännessä informaationäytössä näytettävään tietoon. Tällaisessa tapauksessa valitusta ikonista lähetetään kuitenkin tieto käyttöliittymälle ja kontrolliyksikkö muistaa vain, että kyseinen ikoni on valittu.

5.1.2 CAN-kuuntelijan konfigurointi

CAN-kuuntelijan konfiguraatiotiedostossa listataan ne PGN:t (J1939-parametriryhmä), joita halutaan väylältä kuunnella. Jokaisen PGN:n osalta määritellään mitkä sen parametreista (SPN) halutaan lukea sekä niiden sijainti ja pituus paketin dataosiossa. Näiden lisäksi jokaiselle SPN:lle on lueteltu ne tilat, joissa niitä käytetään. Alla on esitetty CAN-väylältä kuunneltavan J1939-paketin PGN:n konfiguraatio.

```
pgn:
        - 61444: # Electronic Engine Controller 1
                - spn:      190         # RPM
                  offset:   3           # Offset of j1939 data
                  length:   2           # Length of spn
                  mode:     [RPM]       # State machine mode

        - 65263: # Engine Fluid Level/Pressure 1...
```

Diagnostiikkaviestien osalta merkitään ne SPN:t, joiden ilmaantuminen diagnostiikkaviestissä näytetään näytöillä ja mahdollisesti voidaan myös määritellä arvoalue, jonka sisällä SPN sallitaan. Näytöllä näytettävä diagnostiikkaviesti on konfiguraatiotiedostossa määritelty segmenttiryhmä, jota vilkutetaan virhetilanteen sattuessa. Myös käyttöliittymälle lähtevän virheviestin nimi mainitaan jokaisen SPN:n kohdalla. Alla esimerkki diagnostiikkaviestin konfiguraatiosta.

```
diagnostics:

        - spn: 190                      # RPM spn
          valuemin: 5600                # Optional min value (raw)
          valuemax: 16000               # Optional max value (raw)
          lgroup: [Engine, Rpm Error] # Lamp groups
          error: RPM                    # Error message to ui
```

Informaationäytössä, jossa on 7-segmenttinäytöt ja traktorin tilaan liittyvät ikonit, on tarkoitus näyttää CAN-väylältä saatua tietoa. J1939-paketeissa SPN:ien arvot on usein skaalattu eri tavoin, joten eri tiloille on konfiguraatiotiedostossa määritelty laskukaava, jolla näytettävä arvo lasketaan. Laskukaavassa voidaan käyttää peruslaskutoimitusten lisäksi SPN:ien suoria arvoja, kokonaiskeskiarvoa tai liukuvaa keskiarvoa. Tällä tavalla voidaan helposti hallita näytettävää tietoa ja laskea sellaisia

arvoja, joita ei suoraan ole CAN-väylältä saatavissa. Esimerkiksi traktorin jäljellä olevan toiminta-ajan laskemisessa on käytetty polttoaineen hetkellisen kulutuksen kokonaiskeskiarvoa ja jäljellä olevan polttoaineen määrää. Jokainen tila on nimetty ikonin mukaan, jolloin kontrolliyksikön tilan tunnisteena toimii ikonin indeksi. Alla on esimerkki tilan konfiguraatiosta. Ennen tilojen listausta määritellään aloitustila sekä ikoni, joka on käynnistettäessä valittuna. Tämän jälkeen listataan jokaisen tilan nimi(ikoni), laskukaava, 7-segmenttinäytön nimi, sekä muoto, jossa tieto näytetään. Laskukaavassa SPN:t on ilmaistu edeltävällä $-merkillä.

```
startmode: RPM
startcursor: RPM
modes:
        - mode:      RPM             # State name (icon)
          equation:  $190 * 0.125    # Output value calculation
          dgname:    Info Digits     # Where to display
          format:    nn nn           # Display format
```

5.1.3 UI-kuuntelijan konfigurointi

Käyttöliittymän kuuntelijan konfiguraatiotiedosto sisältää kaikki mahdolliset käskyt, joita voidaan TCP-yhteyden yli antaa. Sen avulla voidaan muokata käskyn ulkomuotoa, paluuviestissä tulevan tiedon sisältöä, diagnostiikkaviestien nimiä ja kuunteluporttia. Käskyn ulkomuodon tai paluuviestissä tulevan tiedon muuttamiselle ei pitäisi olla tarvetta ja ne voitaisiinkin jatkokehityksen kannalta poistaa konfiguraatiotiedostosta. CAN-väylältä kuunneltaville diagnostiikkaviesteille annetaan nimet ja ne saavat järjestyksen perusteella indeksin. Diagnostiikkaviestien nimiä käytetään CAN-kuuntelijan konfiguraatiossa yksilöimään diagnostiikkaviestit. Kun diagnostiikkaviesti vastaanotetaan CAN-väylältä, käyttöliittymälle lähetetään vain indeksi ja tarvittaessa käyttöliittymä voi pyytää listan diagnostiikkaviestien nimistä.

5.2 Rajapinnat

Toteutetun kontrolliyksikön käyttämiä ulkoisia rajapintoja on kolme: toisessa diplomityössä toteutetun näyttöyksikön rajapinta, käyttöliittymän rajapinta ja CAN-väylä. Rajapinnat näyttöyksikölle ja käyttöliittymälle on itse suunniteltu. CAN-väylällä käytetään J1939-protokollaa. Sisäisiä rajapintoja ovat säikeiden välinen viestintä, joka myös toteutettiin itse käyttäen yksinkertaista FIFO-puskuria.

5.2.1 Näyttöyksikkö

Näyttöyksikön rajapinta suunniteltiin mahdollisimman yksinkertaiseksi ja geneeriseksi, jotta sitä olisi mahdollista käyttää vaikka näyttöjen sisältö muuttuisi. Yksinkertaisimmillaan sillä voidaan sytyttää ja sammuttaa yksittäisiä segmenttejä. Joitakin yleisesti käytettyjä ominaisuuksia ja näytettävien kuvioiden toteutuksia siirrettiin kuitenkin tehtäväksi näyttöyksikölle. Näitä olivat mm. 7-segmenttinäyttöjen ohjaus numeroilla, vilkuttaminen, useamman segmentin yhtäaikainen ohjaus, sekä navigointi.

Rajapinta sisältää käskyt, joilla voidaan ohjata yksittäisiä segmenttejä, joukkoja (engl. groups), näyttöjä, ikoneita ja 7-segmenttinäyttöjä. Näiden lisäksi voidaan säätää kirkkautta tai navigoida ikoneiden välillä. Rajapintaan on sisällytetty myös käsky, jolla voidaan asettaa näyttöjen yksittäisiä lamppuja niiden todellisen sijainnin perusteella, jolloin konfiguraatiotiedostoissa käytetty nimeäminen ohitetaan. 7-segmenttinäyttöjen ohjauksessa annetaan konfiguraatiotiedostossa määritelty näytön tunniste ja näytettävä numerosarja tai kirjaimet. Segmenttejä, näyttöjä, ryhmiä tai ikoneita asetettaessa annetaan indeksin lisäksi arvo väliltä 0–255. Arvo 255 tarkoittaa jatkuvaa palamista ja arvo 0 sammutusta. Arvot 1–254 ilmaisevat vilkutustaajuuden sekunnin sadasosissa. Näyttöyksikön rajapinnan käskyt on lueteltu liitteessä C.

Käytännössä rajapintana käytettiin kahta Linuxin tarjoamaa nimettyä FIFO-jonoa kaksisuuntaiseen viestintään. Näyttöyksiköltä voidaan edellyttää vastausta asettamalla käskyn reply-tavu arvoon yksi. Tätä käytetään silloin, kun käyttöliittymältä tulee käsky ja halutaan tietää onnistuiko se. Jos reply-tavun arvo on nolla, näyttöyksikkö ei lähetä vastausta käskyn onnistumisesta. Sen sijaan virhetilanteissa näyttöyksikkö lähettää aina vastauksen. Virheviestit voivat koskea esimerkiksi yksittäistä segmenttiä kokonaista ryhmää asetettaessa, jolloin jokaisesta virheellisestä segmentistä tulee oma viesti.

5.2.2 Käyttöliittymä

Rajapinta käyttöliittymälle on toteutettu TCP-yhteyttä käyttämällä. Kehitysalustalle on asetettu kiinteä IP-osoite ja konfiguraatiossa on määritelty portti, jota kuunnellaan. Tiedonsiirtoprotokollaksi valittiin TCP UDP:n sijasta, sillä tiedonsiirron tulisi olla luotettavaa. Kommunikointiin käyttöliittymän kanssa toteutettiin oma sovellustason protokolla, joka sisältää tarvittavat käskyt näyttöjen ohjaamiseen. Ne mahdollistavat eri tilojen välillä siirtymisen yksinkertaisten next, previous

ja select-käskyjen avulla. Rajapintaan on tuotu myös näyttölasin tulkin rajapinnasta alemman tason käskyt, jolloin tarpeen vaatiessa näyttöjä voidaan hallita täysin TCP-yhteyden yli. Tästä on hyötyä, jos esimerkiksi halutaan kokeilla jotain muuta toimintalogiikkaa tai näyttöjä ohjaavana ohjelmistona halutaan käyttää ennestään olemassa olevaa ohjelmistoa pienin muutoksin.

TCP-yhteyden muodostamiseen käytetään Linux-käyttöjärjestelmän tarjoamaa socket-yhteyttä. Kehitysalustalle on asetettu staattinen IP-osoite ja UI-kuuntelija avaa socketin kuuntelemaan porttia 6542. Porttia voidaan myös tarvittaessa muuttaa UI-konfiguraatiossa. Tämän jälkeen käyttöliittymäpuolen laite ottaa yhteyden kehitysalustaan, jonka jälkeen viestien vastaanotto ja lähetys tapahtuu käyttäen Linuxin tähän tarkoitukseen tarjoamia funktiota.

Protokollalla välitettävät viestit ovat seuraavanlaisia:

```
!Command:parameter1:parameter2;
```

Viestit alkavat huutomerkillä (!) ja päättyvät puolipisteeseen (;). Paketin sisäiset parametrit, joita ovat eri käskyihin liittyvät lisätiedot, erotellaan toisistaan kaksoispisteellä. Parametrien määrä riippuu käskystä. Aloitus- ja lopetusmerkkien käyttö helpottaa käskyjen löytämistä TCP-virrasta, sillä ohjelman tarvitsee vain etsiä aloitusmerkki ja odottaa lopetusmerkin saapumista, jonka jälkeen koko viesti voidaan käsitellä. Kaikista vastaanotetuista käskyistä lähetetään kuittaus, joka kertoo onnistuiko käskyn käsittely ja mahdollisesti käskyn epäonnistuessa lisätietoja siitä, mikä meni pieleen. Myös CAN-väylältä vastaanotetut diagnostiikkaan liittyvät virheviestit välitetään UI-kuuntelijan konfiguraatiossa määritetyillä virheiden indekseillä. Protokollan dokumentaatio kaikista käskyistä löytyy liitteestä A.

5.2.3 CAN-väylä

CAN-väylän rajapinta ohjelmiston kannalta on samanlainen socket-rajapinta kuin käyttöliittymällä. Saatavilla oleva Linuxin CAN-tuki käyttää socket-pohjaista yhteystekniikkaa, jolloin yhteys avataan sopivilla parametreilla ja väylää voidaan kuunnella. J1939-tuki näkyy ohjelmassa vastaanotetun viestin lähettäjän osoitteen tietorakenteessa (struct sockaddr_can), josta löytyy lisäksi PGN, lähettäjän J1939-osoite ja J1939-nimi. Viestien vastaanotto tapahtuu kuten UI-kuuntelijassa hyödyntäen Linuxn tähän tarjoamia funktiota. CAN-väylän kuuntelussa yhteyttä ei tarvitse erikseen muodostaa, vaan viestien vastaanotto onnistuu saman tien. Kehitysalustalle ei ole asetettu J1939-protokollan mukaista osoitetta eikä nimeä, sillä sen on ainoastaan tarkoitus kuunnella väylän liikennettä.

Kehitysalustan fyysinen CAN-rajapinta koostuu kolmesta pinnistä: CAN-High, CAN-Low ja CAN-Ground. CAN-Ground on kehitysalustan omasta maadoituksesta eristetty, ja se on tarkoitettu ainoastaan CAN-väylän käyttöön. Kehitysalustalla on myös mahdollista valita oikosulkupalan avulla käytetäänkö 120 Ohmin päätevastusta CAN-High ja CAN-Low johtimien välillä.

5.2.4 Sisäiset rajapinnat

Ohjelmiston sisäisiä rajapintoja ovat CAN-kuuntelijan ja UI-kuuntelijan viestinvälitysrajapinnat pääsäikeelle. Tähän tarkoitukseen tehtiin yksinkertainen FIFO-jono jokaiselle yhdensuuntaiselle viestinnälle. Sisäisiä FIFO-jonoja tarvitaan tällöin kolme kappaletta: CAN-kuuntelijalta pääsäikeelle, UI-kuuntelijalta pääsäikeelle ja pääsäikeeltä UI-kuuntelijalle. Toteutettu FIFO-jono on säieturvallinen, jolloin luettaessa tai kirjoitettaessa jonoon pääsy rajoitetaan mutex-lukituksella vain yhdelle säikeelle kerrallaan. FIFO-jonon sisältö voi olla mitä tahansa, mutta yhden alkion koko määritellään jonoa luotaessa. CAN-kuuntelijan ja UI-kuuntelijan viestien sisällöt eroavat toisistaan.

CAN-kuuntelijan viestit ovat ohjelmassa 5.1 listatun tietorakenteen mukaisia. Kentät *pgn_ index* ja *spn_ index* viittaavat taulukoihin, jotka on luotu CAN-konfiguraatiotiedoston pohjalta. Kenttä *value* sisältää vastaanotetun SPN:n arvon ja viimeinen *error*-kenttä sisältää tiedon siitä, löytyikö SPN mahdollisesti diagnostiikkaviestistä ja onko virhe vielä aktiivinen. CAN-kuuntelija siis lähettää viestin pääsäikeelle jokaisesta vastaanottamastaan SPN:stä, jotka se on konfiguroitu huomioimaan.

```
/**
 * Struct for a message from/to can_listener
 */
typedef struct can_l_msg_t {
   int pgn_index;    /*!< PGN index inside can info from yaml */
   int spn_index;    /*!< SPN index inside pgn info */
   int value;        /*!< Value of spn */
   char error;       /*!< If this spn was found in
                         diagnostic message
                     * 0 not diagnostic message,
                     * 1 new active
                     * 2 previously active, now deactivated
                     */
} can_l_msg_t;
```

Ohjelma 5.1 *CAN-kuuntelijan pääsäikeelle lähettämien viestien tietorakenne.*

UI-kuuntelijan viestit ovat ohjelmassa 5.2 listatun tietorakenteen mukaisia. Kentät *ui_command*, id, value, digits ja digit_timeout sisältävät tietoa, jonka UI-kuuntelija lähettää pääsäikeelle. Kenttä *ui_command* on tunniste, joka on määritelty UI-konfiguraatiossa ja ohjelmakoodissa kiinteästi. Kentät *id* ja *value* sisältävät segmentin, ryhmän tai ikonin tunnisteen ja asetettavan arvon. Asetettaessa 7-segmenttinäyttöjä kenttään *digits* talletetaan haluttu arvo ja kenttään *digit_timeout* voidaan asettaa aika, jonka aikana mikään muu toiminto ei pääse vaikuttamaan 7-segmenttinäyttöihin. Näin estetään esimerkiksi CAN-väylältä saapuvia viestejä kirjoittamasta yli näytölle manuaalisesti asetettua tekstiä.

Tietorakenne struct reply sisältää tietoa, jonka pääsäie lähettää vastausviestissään UI-kuuntelijalle. Sisällytettävä tieto on käytännössä kontrolliyksikön nykyinen tila sekä vastattavaan käskyn liittyvää tietoa. Tietorakenne struct error on käytössä silloin, jos näytön väliohjelmistolta saapuu virheviesti ja se pitää välittää käyttöliittymälle.

```
/**
 * Struct for a message from/to ui_listener
 */
typedef struct ui_l_msg_t {
   char ui_command;     /*!< UI command id */
   int id;              /*!< ID of a lamp/group/icon */
   int value;           /*!< Value of possible third parameter */
   char digits[9];      /*!< Value for setting digits */
   char digit_timeout;  /*!< Timeout for digit lock */
   struct reply {       /*!< Data to ui_listener */
      char fail;        /*!< If command failed this is 1 */
      int id;           /*!< ID of icon/group/animation */
      char screen;      /*!< ID of current screen */
      int value;        /*!< Value of setting */
      char cur_pos;     /*!< Value of current cursor position */
      char cur_state;   /*!< Value of current state */
   } reply;
   struct error {
      char active;      /*!< Possible error from spi middleware
                             active */
      char message[64]; /*!< Error message from spi middleware */
   } error;
} ui_l_msg_t;
```

Ohjelma 5.2 *UI-kuuntelijan pääsäikeelle lähettämien ja vastaanottamien viestien tietorakenne.*

5.3 Ohjelmiston rakenne

Rakenteeltaan kontrolliyksikkö sisältää kolme säiettä. Säikeitä on yksi CAN-väylälle, yksi käyttöliittymän kuunteluun ja kolmas säie eli pääsäie, joka toimii tilakoneena. Toteutukseen käytettiin Pthreads-kirjastoa [15], joka on POSIX-standardin mukainen C/C++-kielelle tehty kirjasto. Se sisältää säikeiden käsittelyyn tarvittavat funktiot ja muita rinnakkaisohjelmoinnissa tarvittavia toimintoja, kuten mutex-lukituksen. Säikeiden välinen kommunikointi tapahtui aliluvussa 5.2.4 kuvattujen rajapintojen avulla. Pääsäie, eli tässä tapauksessa tilakone, luo tarvittavat FIFO-jonot sisäisille rajapinnoille ja välittää osoittimen näistä eri säikeille niiden käynnistyksen yhteydessä. Pääsäie luo myös muut säikeet ja lukee konfiguraatiotiedostot.

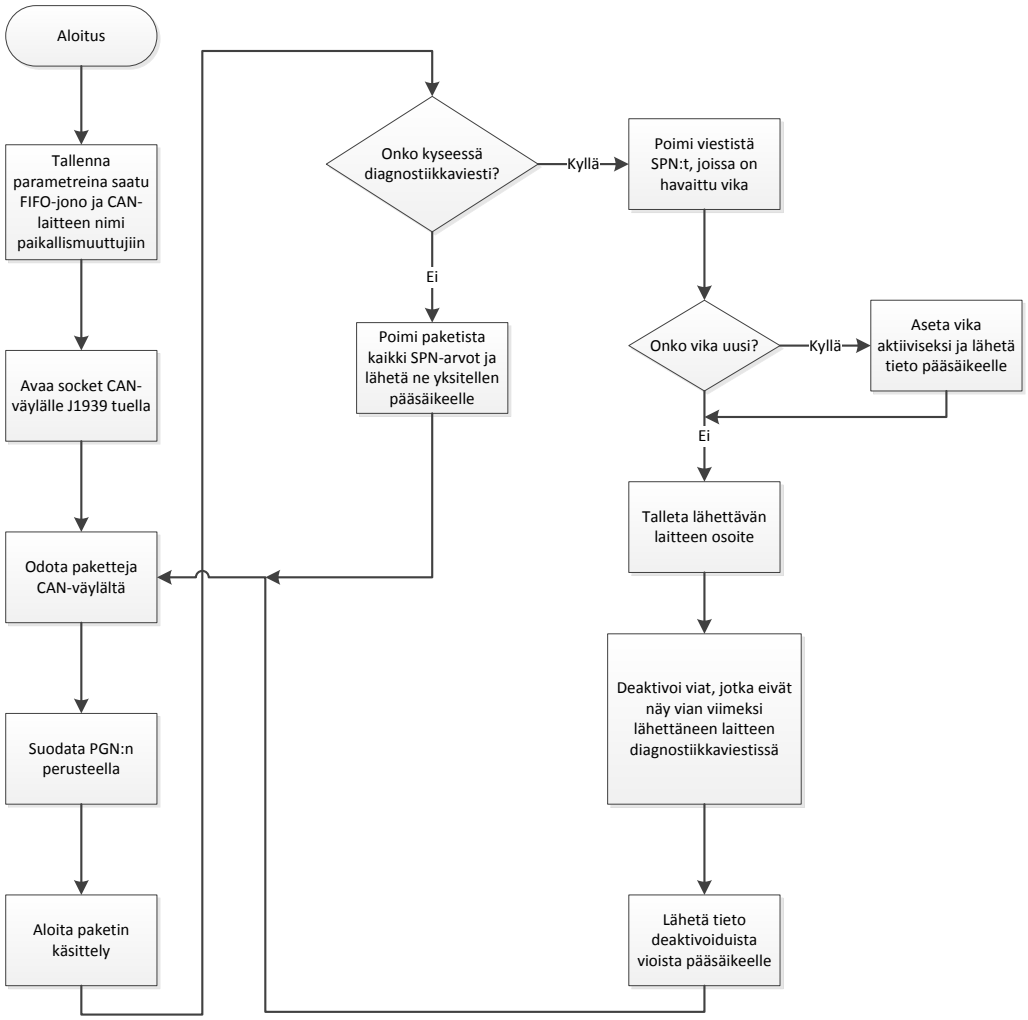
5.3.1 CAN-väylän kuuntelija

CAN-väylän kuuntelija on toteutettu säieturvallisesti siten, että niitä voidaan luoda useampia esimerkiksi jokaiselle CAN-väylälle erikseen. Säiekohtaiset muuttujat on tallennettu tietorakenteeseen, jota kuljetetaan kutsuttavien funktioiden välillä. Jokainen säie alustaa oman tietorakenteensa käynnistyessään ja tuhoaa sen sulkeutuessaan.

CAN-väylän kuuntelijan toimintaperiaate on kuvan 5.1 mukainen. Käynnistyessään se avaa tarvittavan socketin CAN-väylälle sopivilla asetuksilla. Tässä vaiheessa asetetaan myös J1939-protokollan tuki, jolloin tavallisten CAN-viestien kuuntelu samalla socketilla ei ole mahdollista. Lisäksi kehitysalustalla on kaksi CAN-oheislaitetta: fyysinen ja virtuaalinen CAN, joista toinen valitaan konfiguraatiotiedoston perusteella.

Alkurutiinien jälkeen säie siirtyy ikuiseen silmukkaan, jossa se jää odottamaan Linuxin tarjoamassa odottavassa *rcvfrom*-funktiossa [25] viestejä CAN-väylältä. Viestin tultua suoritetaan ensimmäisenä suodatus PGN:n perusteella. Jos PGN löytyy kuunneltavien PGN:n listalta konfiguraatiosta, se käsitellään. Muussa tapauksessa vastaanotettua viestiä ei käsitellä, vaan siirrytään odottamaan seuraavaa viestiä. Vastaanotettuaan suodatuksen läpäisevän viestin säie poimii siitä konfiguraatiotiedostoon asetetut SPN:t niiden sijainnin ja pituuden mukaan. Poimittuaan SPN:n se lähettää sen pääsäikeelle aliluvussa 5.2.4 esitellyn rajapinnan mukaisesti.

Diagnostiikkaviestit käsitellään eri tavoin, sillä niiden sisältö poikkeaa tavanomaisesta J1939-viestistä. J1939-protokollan mukainen DM1-diagnostiikkaviesti sisältää kaikki aktiiviset diagnostiikkakoodit. Viestin dataosiossa on listattu kaikki sellaiset SPN:t, joissa on havaittu jokin vika. Jokaisen SPN:n kohdalla on tarkempi kuvaus



Kuva 5.1 *CAN-kuuntelijan toimintaperiaate.*

vian tyypistä (esimerkiksi raja-arvo ylitetty, virheellinen arvo) sekä tieto siitä, kuinka monta kertaa vika on toistunut.

CAN-kuuntelija poimii näistä tiedoista ainoastaan SPN:t ja vertaa niitä konfiguraatiotiedostossa lueteltuihin. Jos jokin luetelluista SPN:stä löytyy diagnostiikkaviestistä, CAN-kuuntelija lähettää tästä tiedon pääsäikeelle. CAN-kuuntelija pitää kirjaa aktiivisista diagnostiikkakoodeista ja lähettää pääsäikeelle viestin ainoastaan muutoksen tapahtuessa. Vikakoodi deaktivoituu, kun sitä ei enää löydy diagnostiikkaviestistä. On otettava huomioon, että diagnostiikkaviestin voi lähettää usea CAN-väylälle liitetty laite, jolloin deaktivoitaessa vikakoodia ainoastaan laite, joka sen viimeisimpänä aktivoi, voi sen deaktivoida. Lähettävä laite tunnistetaan sen

osoitteesta ja näin vältytään siltä, että kunnossa oleva laite deaktivoisi jonkin muun laitteen vikakoodin. CAN-kuuntelija jatkaa ikuisessa silmukassa viestien käsittelyä, kunnes ohjelma suljetaan.

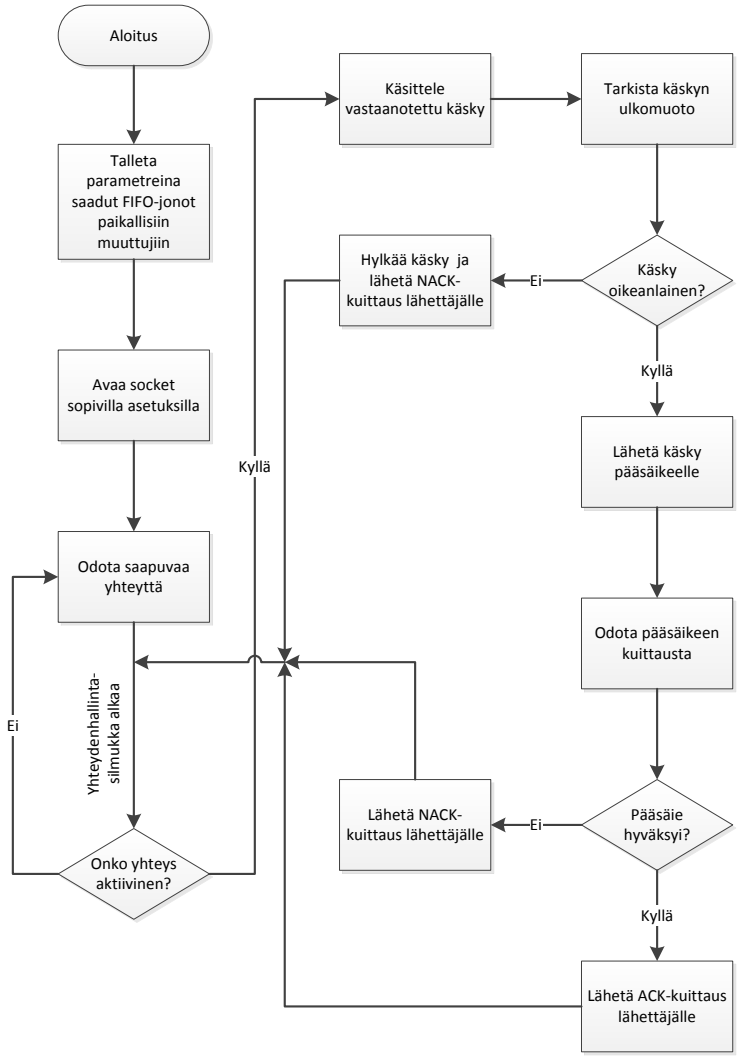
5.3.2 UI-kuuntelija

TCP-yhteyttä kuunteleva käyttöliittymän kuuntelija muistuttaa toimintaperiaatteeltaan CAN-kuuntelijaa. Mainittavana erona on kuitenkin se, että sen molemmat rajapinnat käyttöliittymälle ja pääsäikeelle ovat kaksisuuntaisia. Toimintaperiaate on kuvan 5.2 mukainen.

Käynnistyessään UI-kuuntelija avaa socketin TCP-yhteydelle porttiin, joka on mainittu konfiguraatiotiedostossa. Socketin luomisen jälkeen se jää kuuntelemaan uusia yhteyksiä Linuxin tarjoamalla odottavalla *accept*-funktiolla. Uuden yhteyden muodostuttua siirrytään yhteydenhallintasilmukkaan, jossa viestejä vastaanotetaan ja lähetetään. Myös pääsäikeeltä saadut viestit vastaanotetaan ja lähetetään tässä. Huomioitavaa on, että tässä tapauksessa TCP-yhteyden vastaanotossa käytetään Linuxin tarjoamaa *rcv*-funktiota ei-odottavana, jotta pääsäikeeltä saadut viestit tulevat käsiteltyä. TCP-yhteyksiä sallitaan vain yksi kerrallaan epäselvyyksien välttämiseksi.

Vastaanotettuaan viestin käyttöliittymältä UI-kuuntelija tarkastaa käskyn oikean muodon. Tämä tapahtuu käyttäen *sscanf*-funktiota, joka pilkkoo viestin osiin *format*-merkkijonon perusteella ja tallentaa löytämänsä arvot muuttujiin [33]. Jos viesti oli oikeanmuotoinen, siirrytään tarkastamaan sen sisältöä. UI-kuuntelija tarkastaa, että annetun käskyn parametrit ovat sallituissa rajoissa ja että ohjattava segmentti, ryhmä tai ikoni löytyy näyttöjen konfiguraatiosta. Tämän jälkeen viesti talletetaan aliluvussa 5.2.4 esitettyyn muotoon ja lähetetään pääsäikeelle. Jos viesti sisältää vain pyynnön esimerkiksi ikoneista, UI-kuuntelija vastaa itse lähettämällä listan pyydetyistä tiedoista eikä pääsäikeelle välitetä mitään tietoa. Viestin käsittelyn jälkeen palataan yhteydenhallintasilmukkaan.

Lähettäessään TCP-viestejä käyttöliittymälle UI-kuuntelija lisää ne jonoon, joka käsitellään yhteydenhallintasilmukassa viestin käsittelyn jälkeen. Jos jonossa on lähetettäviä viestejä, koko jono tyhjennetään tässä vaiheessa. Seuraavaksi yhteydenhallintasilmukassa vastaanotetaan pääsäikeeltä saadut viestit ja hoidetaan niiden käsittely. Pääsäikeeltä saadut viestit ovat enimmäkseen kuittausviestejä, joista selviää onko annettu käsky onnistunut. Onnistuneesta tai epäonnistuneesta käskystä lähetetään aina kuittaus takaisin käyttöliittymälle. CAN-väylältä saapuvat diagnostiikkaviestit ja näyttöyksikön virheviestit välitetään myös käyttöliittymälle. Vastaanotet-



Kuva 5.2 *UI-kuuntelijan toimintaperiaate.*

tuaan viestit pääsäikeeltä yhteydenhallintasilmukka palaa lukemaan TCP-yhteyden viestejä.

Jos lähetettävänä tai vastaanotettavana ei ole viestejä, yhteydenhallintasilmukka nukkuu millisekunnin ajan, jotta säie ei turhaan kuluttaisi suoritinaikaa. Yhteydenhallintasilmukasta poistutaan takaisin odottamaan uutta yhteyttä, kun yhteys ei ole enää aktiivinen, ja säie tuhotaan ainoastaan ohjelman sulkeutuessa.

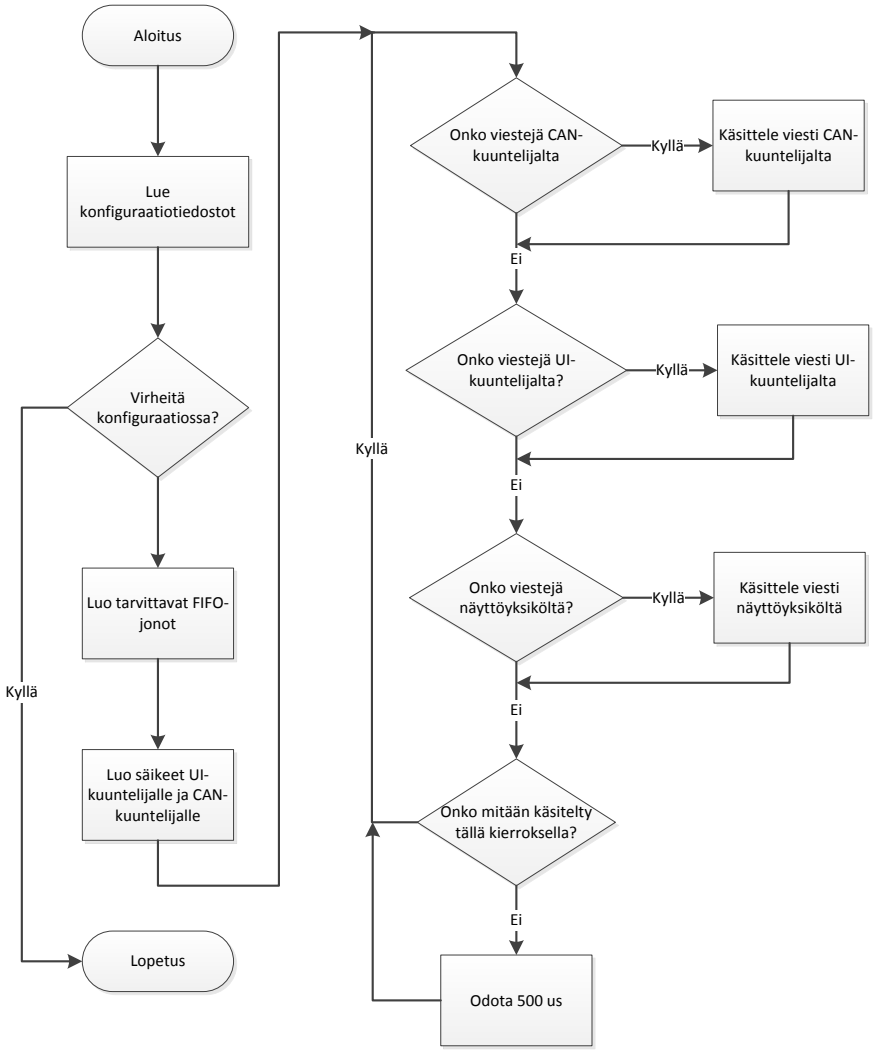
5.3.3 Pääsäie

Pääsäie toimii tilakoneena, joka vastaanottaa ja lähettää viestejä UI-kuuntelijalle, näyttöyksikölle ja CAN-kuuntelijalle. Tämä säie toimii koko prosessin pääsäikeenä, joten sen tehtäviin kuuluu tilakoneena toimimisen lisäksi muiden säikeiden luominen, komentoriviparametrien käsittely ja konfiguraatiotiedoston lukeminen. Komentoriviparametrien tulkinta, muiden säikeiden luominen ja käynnistäminen sekä niihin liittyvien FIFO-jonojen luominen suoritetaan ennen varsinaiseen silmukkaan siirtymistä. FIFO-jonojen toteutus on vaikeusasteeltaan melko triviaalia, joten niitä ei käsitellä tässä yhteydessä. Konfiguraatiotiedoston lukeminen on kuitenkin tarkemmin esitetty aliluvussa 5.3.4. Ennen varsinaiseen tilakoneeseen siirtymistä avataan FIFO-jonot näyttöyksikölle.

Alkurutiinien jälkeen siirrytään silmukkaan, jossa luetaan viestejä vuorotellen eri jonoista. Kontrolliyksikön toiminta on esitetty kuvassa 5.3.

CAN-kuuntelijan konfiguraatiotiedostossa on jokaiselle tilalle määritelty laskukaava. Saatuaan viestin CAN-kuuntelijalta tilakone hakee nykyisen tilan mukaisen laskukaavan konfiguraatiosta ja laskee sen arvon. Lopputulos näytetään konfiguraatiotiedoston mukaisessa muodossa halutulla lukunäytöllä. Laskukaavan purkuun käytetään Dijkstran kehittämää shunting-yard-algoritmia [24], jossa infix-notaatio muunnetaan ensin postfix-muotoon ja lasketaan tämän jälkeen pinon avulla. Tähän tarkoitukseen löytyi valmiina yksinkertainen toteutus, johon pienillä muutoksilla voitiin muuttujina käyttää SPN-arvoja, keskiarvoa ja liukuvaa keskiarvoa [20].

Projektin loppuvaiheessa päätettiin käyttöliittymän puolelle asentaa rattiohjain sekä polkimet. Polkimien asennolla haluttiin olevan vaikutus näytöillä näytettäviin tietoihin, joten tähän tarkoitukseen käyttöliittymän rajapintaan lisättiin käsky, jolla voidaan asettaa SPN:n arvo manuaalisesti. Tilakone lisää UI-kuuntelijalta saadun SPN:n arvon CAN-kuuntelijalta saapuvaan jonoon, jolloin SPN näyttää saapuvan CAN-väylältä. Tällä tavalla voidaan hyödyntää samaa ohjelmistorakennetta ja riittää, kun konfiguraatiotiedostossa yhden tilan laskukaavaan sisällytetään käyttöliittymän asettama SPN. Käyttöliittymä ei lähetä polkimen asentoa jatkuvasti, vaan ainoastaan arvot 0, 50 ja 100 asennon muuttuessa. Tätä puutetta korjaamaan tilakone lähettää käyttöliittymän asettamaa SPN-arvoa tasaisin väliajoin CAN-kuuntelijan jonoon, jotta liukuvaa keskiarvoa hyödyntäen saadaan esimerkiksi sulavasti liikkuva nopeusnäyttö.



Kuva 5.3 *Kontrolliyksikön toimintaperiaate.*

5.3.4 Konfiguraation luku

Konfiguraatiotiedostojen lukeminen tapahtuu pääsäikeessä ohjelman alussa. Lukuun käytetään kirjastoa *libyaml* [27], joka on YAML-tiedostojen lukuun tarkoitettu C-kielen kirjasto. Se tarjoaa rajapinnan, jolla tiedosto voidaan lukea alusta loppuun sekventiaalisesti. Lukeminen perustuu tapahtumiin (engl. event), joita saadaan paluuarvona kirjaston funktiosta *yaml_parser_parse*. Jokainen tapahtuma sisältää jonkin tietorakenteen arvon tai tiedoston rakenteeseen liittyvän tapahtuman. Esimerkiksi listan alku ilmaistaan tapahtumalla *YAML_SEQUENCE_START_EVENT*, skalaarin arvo tapahtumalla *YAML_SCALAR_EVENT* ja tiedoston loppu tapahtumalla *YAML_STREAM_END_EVENT*.

Ohjelmiston toiminta konfiguraatiotiedoston luvussa voidaan kuvata silmukkana, jossa kutsutaan *yaml_parser_parse*-funktiota ja toimitaan paluuarvon tapahtuman mukaan. Käytännössä ohjelmakoodi on jaettu useampaan funktioon selkeyden vuoksi siten, että esimerkiksi jokaiselle konfiguraatiotiedostolle on oma käsittelyfunktio ja jokainen lista käsitellään omassa funktiossaan.

5.3.5 Tukitoiminnot

Tukitoiminnoilla tässä tapauksessa tarkoitetaan pääasiassa toimintoja automatisoivia skriptejä sekä oheisohjelmistoja, joista tärkeimpiä ovat GDB-debuggeri ja SSH-palvelin. Kehitysalustan mukana toimitettu Linux-versio sisältää Busybox-ohjelmiston, joka on työkalupaketti Linuxin yleisimmistä komennoista. Se on erityisesti suunniteltu sellaisiin järjestelmiin, joissa resurssit ovat rajalliset ja tästä johtuen osa työkaluista on alkuperäisistä riisuttuja versioita. SSH-palvelin löytyi entuudestaan, eikä se vaatinut toimiakseen konfigurointia. Sen sijaan GDB-debuggeria ei löytynyt eikä sopivaa versiota iproute2-työkalusta, jota tarvittiin CAN-laitteen konfiguroinnissa.

GDB-debuggerin ristikäännös lähdekoodista sujui pääsääntöisesti ongelmitta. Joitakin muutoksia täytyi tehdä konfiguraatiotiedostoihin ja yksi riippuvuuksiin kuuluvista kirjastoista kääntää erikseen. Käännöksen jälkeen tiedostot voitiin kopioida kehitysalustalle käytettäväksi.

Iproute2 on Linuxin työkalukokoelma, jolla voidaan muuttaa verkkoasetuksia. Se on suunniteltu korvaamaan monia aiempia työkaluja, kuten *ifconfig*, *route* ja *arp*. Syynä uuden työkalun kehittämiseen on, että esimerkiksi *ifconfig* voi käyttäytyä arvaamattomasti uusimmissa ytimen versioissa, eivätkä vanhat työkalut tarjoa verkkoliikenteen ohjausta tai priorisointia[36]. Tässä työssä käytettiin J1939-protokollan tuen sisältävää versiota iproute2-ohjelmistosta, jolla CAN-laitteelle voitiin kytkeä J1939-tuki päälle. Kääntäessä jouduttiin poistamaan joitakin arp-ominaisuuksia niiden riippuvuuksien puuttumisen vuoksi, mutta niitä ei olisi muutenkaan käytetty tässä projektissa. Tämän lisäksi lähdekoodista jouduttiin muokkaamaan yhden funktion nimeä, sillä samanniminen funktio löytyi jo Linux-ytimen otsikkotiedostoista.

Lisäksi CAN-väylän toimivuuden testaukseen käytettiin *canutils*-työkalukokoelmaa. Se sisältää työkalut, joilla voidaan lähettää ja tarkkailla CAN-väylän liikennettä. Tämä osoittautui hyödylliseksi testattaessa CAN-kuuntelijan viestien vastaanoton oikeellisuutta. Tässä projektissa käytetyt työkalut sisälsivät lisäksi tuen J1939-protokollan viesteille.

Skriptejä käytettiin automatisoimaan osa toiminnoista. Esimerkiksi CAN-laitteen

asetus iproute2-työkaluja käyttäen tehtiin skriptin avulla. Laitteen käynnistyksen yhteydessä ajetaan myös skripti, joka asentaa näyttölaitteen ajurin, CAN-laitteen konfiguroinnin sekä käynnistää ohjelmistot ja CAN-nauhoitteen. Näin käyttäjän ei tarvitse kuin laittaa virrat päälle ja näyttölaite on toimintavalmis.

5.4 Testaus

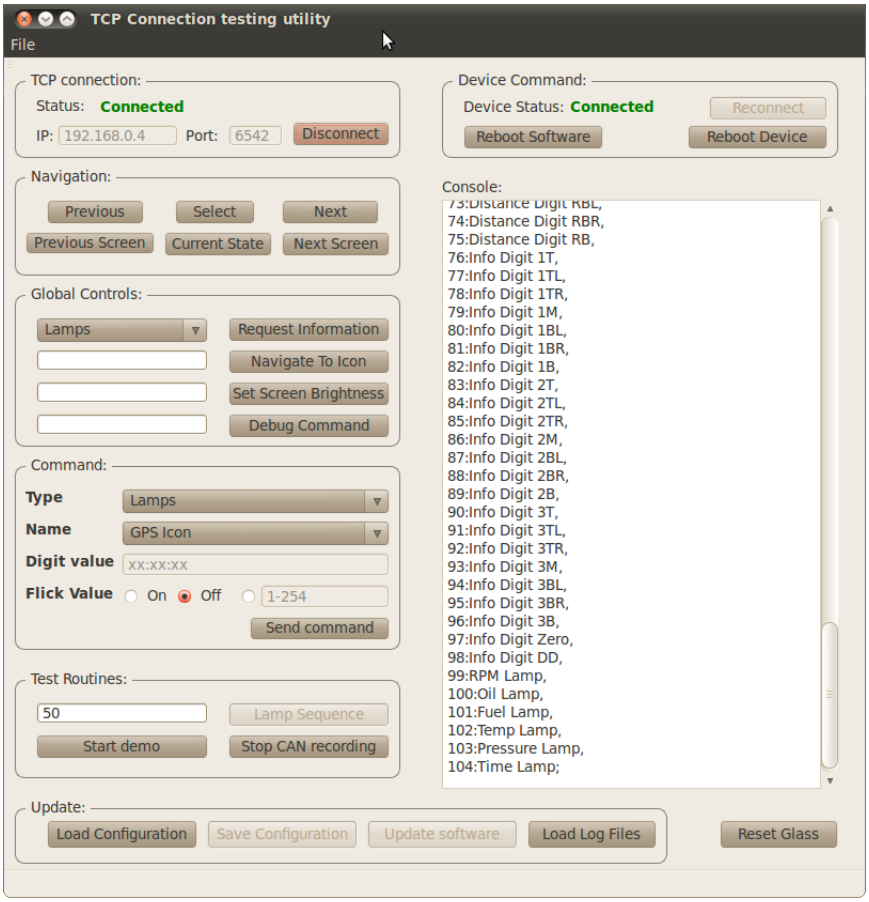
Kuten missä tahansa ohjelmistoprojektissa testaus on olennainen osa projektia, jolla varmistetaan, että ohjelmisto vastaa toiminnaltaan haluttua. Testausta suoritettiin jatkuvasti ohjelmiston ominaisuuksia lisättäessä ja eri moduuleja yhteen sovitettaessa. Pääosin testaus tehtiin käsin, eikä automaatiota juurikaan käytetty lukuun ottamatta joitakin CAN-kuuntelijan testausmenetelmiä. Vaikka automaatioita ei käytetty, pyrittiin testitapaukset valitsemaan siten, että oikealla arvoalueella toiminta on haluttua ja raja-arvot ylitettäessä virhetilanteet siedetään ohjelman kaatumatta siten, että mitään määrittämätöntä ei tehdä. Testaukseen tehtiin myös TCP-rajapintaan työkalu, joka myöhemmin muutettiin koko järjestelmän hallintatyökaluksi.

5.4.1 Rajapinnat

Ulkoisten rajapintojen testausta helpottamaan tehtiin työkalut sekä CAN-väylälle että TCP-yhteydelle. Työkalut eivät ole täysin automatisoituja, vaan ne vaativat käyttäjältä syötteitä, jotka lähetetään ohjelmistolle rajapinnan kautta. Esimerkiksi TCP-testaustyökalulle täytyy syöttää komennon parametrit, mutta itse komentoa ja käskyn muotoa ei tarvitse kirjoittaa aina uudelleen, mikä nopeuttaa testausta.

TCP-testaustyökalu kehitettiin käyttäen graafisten käyttöliittymien kehitysympäristöä Qt(versio 4.8). Työkalu sisältää kaikki käyttöliittymärajapinnan komennot ja niitä voidaan lähettää yksi kerrallaan ohjelmistolle. Koska kehitysympäristönä on käytetty Qt:ä voidaan työkalu periaatteessa ajaa missä tahansa Qt:ä pyörittävässä ympäristössä. Myöhemmin testaustyökalusta kehitettiin koko ohjelmiston hallintaan tarkoitettu työkalu, jolla voidaan myös päivittää ohjelmisto ja konfiguraatiotiedostot sekä aloittaa ja pysäyttää CAN-nauhoite. Kuvassa 5.4 on esitetty testaustyökalun käyttöliittymä.

CAN-rajapinnan testauksessa hyödynnettiin toista kehitysalustaa, joka lähetti fyysiselle CAN-väylälle viestejä. Viestien lähetyksestä vastasi alkuvaiheessa itse tehty ohjelma, joka tuotti J1939-protokollan mukaisia satunnaisia SPN-arvoja CAN-väylälle, joita CAN-kuuntelija kykeni tulkitsemaan. Tätä käytettiin ennen CAN-nauhoitteen



Kuva 5.4 *Testaustyökalun käyttöliittymä.*

saapumista CAN-kuuntelijan testaukseen. Myös virtuaalista CAN-laitetta hyödynnettiin testauksessa ja lopulta myös valmiissa demolaitteessa CAN-nauhoitteen toistamisessa. Kuvassa 5.4 on esitetty testaustyökalun käyttöliittymä. CAN-rajapinnan testauksessa virheitä löytyi melko vähän ja suurin osa liittyi CAN-laitteen konfigurointiin. Esimerkiksi jos J1939-osoitetta ei oltu asetettu, viestit eivät välittyneet lainkaan. Yllättävän usein toimimattomuuden syynä oli katkos fyysisessä yhteydessä, joka johtui huonosta kontaktista kehitysalustojen välillä hyppylangoilla toteutetussa CAN-väylässä.

Rajapintaa näyttöyksikölle testattiin käyttäen TCP-testaustyökalua, sillä lähes samat käskyt löytyvät TCP-rajapinnasta ja näyttöyksikön rajapinnasta. Samalla voitiin testata viestin kulku käyttöliittymältä aina näyttölasille asti debug-tulostuksia hyödyntäen. Löytämistämme virheistä suurin osa paikannettiin tällä tavalla. Myös sisäisten rajapintojen testaus suoritettiin käyttäen syötteitä ulkoisista rajapinnoista ja tarkkailemalla sisäisen liikenteen oikeellisuutta.

SPI-väylän testaus oli hieman haasteellisempaa, sillä Lumineqin ohjaimiin ei ollut

pääsyä minkäänlaisen debug-liitynnän kautta, eikä ohjelmistoa ollut saatavilla. Ainoaksi vaihtoehdoksi jäi tarkkailla väylää oskilloskoopin avulla. Käytössä olleessa oskilloskoopissa oli digitaaliset ja analogiset sisääntulot. Digitaalisilla sisääntuloilla voitiin kätevästi tarkkailla väylälle lähetettävää dataa ja ajoituksia. Analogisilla sisääntuloilla pystyttiin katsomaan signaalin todellista muotoa ja kuinka esimerkiksi liian pitkät johdot heikensivät sitä tai aiheuttivat siihen häiriöitä.

5.4.2 Ohjelmisto

Ohjelmiston testauksessa käytettiin paljon samoja menetelmiä kuin rajapintojen testauksessa. Syötteitä annettiin rajapintojen kautta ja ohjelman sisäistä toimintaa tarkkailtiin tulosteiden ja debuggaustyökalujen avulla. GDB osoittautui tärkeäksi työkaluksi etsittäessä syitä ohjelmiston virheelliseen toimintaan, ja ilman sitä kehityksestä olisi tullut huomattavasti vaikeampaa. Tosin GDB:stä ei juurikaan ollut apua sellaisissa tapauksissa, joissa ohjelman kaatuminen tapahtui ulkoisessa kirjastossa. Tällöin ainoaksi tavaksi jäi tarkistaa koodi käsin ja ajaa se askel kerrallaan.

Muistivuotojen ehkäisyyn ja tarkkailuun oli tarkoitus käyttää valgrind-ohjelmaa.Valgrind on tarkoitettu ohjelman muistinhallinnan tarkkailuun, ja sillä on mahdollista löytää muistivuodot ja rinnakkaisuuteen liittyvät ongelmat. Valgrind saatiin pienillä muutoksilla konfiguraatiotiedostoihin onnistuneesti käännettyä kehitysalustalle, mutta sen ajaminen kehitysalustalla ei onnistunut, sillä se olisi vaatinut ei-riisutun (engl. non-stripped) version glibc-kirjastosta. Koko glibc-kirjaston kääntäminen uudelleen olisi vaatinut liian paljon aikaa, joten päädyimme muihin ratkaisuihin muistivuotojen etsimisessä. Ohjelma käynnistettiin ja sen muistinkäyttöä tarkkailtiin yksinkertaisesti resurssienhallintaohjelmalla top. Ohjelmaa ajettiin esimerkiksi yön yli ja muistinkulutuksen kasvusta voitiin päätellä, että jossain on muistivuoto. Muistivuodon paikallistaminen tapahtui käymällä koodista läpi muistinvaraukset ja niiden vapautukset. Tämä tapa ei ole aukoton, mutta pahimmat muistivuodot pystyttiin näin havaitsemaan.

Prosessorin käyttöasteen tarkkailuun käytettiin niin ikään top-ohjelmaa. Aiemmin mainitut ikuiset silmukat CAN-kuuntelijassa ja pääsäikeessä olisivat vieneet huomattavan määrän prosessoriaikaa ilman niiden nukutusta. Näin saatiin prosessorin käyttöaste putoamaan noin 80 prosentista 10–20 prosenttiin.

Testausmenetelmänä käytettiin myös koodikatselmuksia, joissa koodia käytiin yhdessä läpi tekijän johdolla. Näistä oli paljon apua itselle huomaamattomien virheiden löytämisessä sekä koodin luettavuuden parantamisessa. Koodikatselmuksissa sai myös uusia näkökulmia erilaisten ongelmien ratkaisemiseen, esimerkiksi itse

toteutettuun monimutkaiseen ratkaisuun saattoi löytyä paljon yksinkertaisempikin ratkaisu.

5.4.3 CAN-nauhoite

Pilkingtonilta saatu CAN-nauhoite on nauhoitettu traktorista käytön aikana. Se sisältää suurimman osan arvoista, joita on tarkoitus näyttää näytöillä. Joitakin arvoja sen sijaan ei ollut saatavilla, kuten öljyn lämpötila ja hydrauliikkatiedot. Tästä huolimatta nauhoitteesta oli suuresti apua mm. ajoituksien selvittämisessä ja sitä käytettiin prototyypissä tuottamaan näytettävää tietoa. Ajoitusten kannalta ongelmia ei ollut ja laitteisto pysyi hyvin mukana viestien vastaanotossa.

Nauhoite on tuotettu käyttämällä CANalyzer-ohjelmaa [39], joka luo tekstitiedoston, jossa jokaisella rivillä on aikaleimalla varustettu yksi J1939-viesti. Nauhoite on noin 9 minuutin mittainen, jonka aikana traktori käynnistetään, sillä ajetaan hetki ja lopuksi se sammutetaan. Toistamiseen ei ollut olemassa valmista työkalua, joten siihen tarkoitukseen tehtiin sellainen. Ohjelma lukee tiedoston rivi kerrallaan, poimii aikaleiman sekä viestin sisällön ja lähettää sen oikeaan aikaan. Toisto-ohjelmaa ajettiin sekä toisella kehitysalustalla oikean CAN-väylän ylitse että virtuaalisen CAN-laitteen avulla varsinaisella kehitysalustalla. Molemmissa tapauksissa viestit menivät perille ja CAN-kuuntelija tulkitsi ne oikein.

Diagnostiikkaviestien osalta testaus oli ongelmallisempaa. Nauhoitteessa ei ole juurikaan diagnostiikkaviestejä, sillä traktori on ollut ajettaessa kunnossa. Diagnostiikkaviestien testaus jäikin J1939-dokumentaation ja käsin luotujen viestien varaan. Aliluvussa 5.4.1 mainittu satunnaisia arvoja CAN-väylälle lähettävä ohjelma tuotti myös satunnaisia diagnostiikkaviestejä tähän tarkoitukseen.

Nauhoite oli tärkeä osa prototyyppilaitteen demonstrointia, joten se asetettiin lopuksi myös käynnistymään automaattisesti virtuaalista CAN-laitetta hyödyntäen.

5.4.4 Järjestelmätestaus

Koko järjestelmän toiminta, mukaan lukien Tampereen yliopiston käyttöliittymä, testattiin Pilkingtonin tiloissa oikeaan traktorin lasiin sijoitetuilla näytöillä. Testausta yhdessä käyttöliittymän kanssa oli tehty jo aiemmin TTY:n tiloissa ja laitteiden välinen kommunikointi oli saatu toimimaan. Järjestelmän toimintakuntoon saattaminen oli yllättävän helppoa, eikä vastoinkäymisiä juurikaan ollut. Käyttöliittymän protokollaan lisättiin rivinvaihto vastauksien perään ja joitakin konfiguraa-

tiotiedostojen virheitä jouduttiin korjaamaan. Käyttöliittymältä saadut navigointikäskyt toimivat halutulla tavalla ja CAN-väylältä poimittujen tietojen näyttäminen onnistui hyvin.

5.5 Dokumentaatio

Dokumentaatiota tehtiin projektin jokaisesta osa-alueesta. Dokumentoinnin tuottamiseen käytettiin yksinkertaista tekstieditoria sekä myöhemmin virallisempia dokumentteja tehtäessä toimisto-ohjelmistoja. Dokumentaatiota tuotettiin projektin edetessä sitä mukaa kun dokumentoitavaa materiaalia tuotettiin.

Seuraavista osa-alueista on tehty dokumentaatio: kehitysympäristön ja työkalujen asennus, YAML-konfiguraatio, käyttöliittymän rajapinta, näyttölasin tulkin rajapinta, kehitysalustan skriptit ja ohjelmiston toiminta. Ohjelmiston tarkempi toiminta on dokumentoitu kattavalla kommentoinnilla käyttäen Doxygen-työkalua [13]. Doxygen luo html-pohjaisen dokumentaation ohjelmakoodin kommenttiosioiden perusteella. Sen avulla käytetyt tietorakenteet ovat helposti saatavilla ja jokaisesta funktiosta on olemassa lyhyt esittely parametreineen. Näiden lisäksi voidaan luoda funktioiden kutsukaavioita ohjelmakoodin ymmärtämisen helpottamiseksi. Tekijän kokemuksen perusteella Doxygen on osoittautunut käteväksi työkaluksi myös silloin, kun ennestään tuntemattoman koodin toimintaa yritetään selvittää.

6. TULOKSET JA NIIDEN TARKASTELU

Tässä luvussa esitellään ja tarkastellaan tuloksia. Messuilla esiteltävä prototyyppi esitellään aliluvussa 6.1. Jatkokehityksen kannalta havaitut puutteet ja ideat käsitellään aliluvussa 6.2.

6.1 Prototyyppi

Kokonaisuutena messuilla esiteltävä laite koostuu useasta osasta, jotka on rakennettu traktorin tuulilasin ympärille. Tuulilasiin laminoitua neljää näyttöä ohjaavat kehitysalusta ja sen ohjelmisto. Tähän ohjelmistoon yhteyden ottaa käyttöliittymän laitteistoa ohjaava kannettava tietokone. Käyttöliittymä koostuu mm. ratista, polkimista, Leap Motion -sensorista ja haptisesta palautteesta.

Messuille tarkoitettu prototyyppi oli toteutukseltaan onnistunut ja se toimi niin kuin sen oli tarkoituskin. Lopulliseksi tuotteeksi sitä ei kuitenkaan voi sanoa, mutta se toimii hyvänä pohjana tuleville tuotteille. Ohjelmisto saatiin toimimaan ja testattua riittävän vakaaksi, eikä esimerkiksi yön yli ajoon jätetty laite ollut aamulla kaatunut.

Prototyypissä on myös joitakin ominaisuuksia tai rajoitteita, jotka eivät välttämättä sovellu lopulliseen tuotteeseen. Esimerkiksi käyttöliittymältä saapuvia TCP-yhteyksiä sallitaan vain yksi kerrallaan. Mikäli usea käyttöliittymän laite haluaisi ohjata näyttöjä samanaikaisesti, se ei olisi mahdollista. Mahdollisuus muuttaa SPN-arvoja TCP-yhteyden kautta ei myöskään ole tarpeellinen ominaisuus, sillä CAN-väylältä saatuja arvoja ei pitäisi olla tarvetta muuttaa. Suurin ongelma prototyypissä on kuitenkin SPI-väylän käyttö näyttöjen ohjaamiseen, sillä se ei sovellu pitkille etäisyyksille sellaisenaan. Tähän ja muihin puutteisiin liittyen jatkokehitysideoita esitellään aliluvussa 6.2.

6.2 Jatkokehitys

Prototyypin valmistuksessa tuli ilmi useita seikkoja, jotka vaativat jatkokehitystä mikäli on tarkoitus tehdä oikea tuote. Seuraavissa aliluvuissa on esitelty joitakin ilmi tulleita ideoita ja ratkaisuja.

6.2.1 SPI-väylän korvaaminen

Jatkokehityksen kannalta tärkeintä olisi päästä eroon SPI-väylästä näyttöjen ohjauksessa. Pitkillä etäisyyksillä se ei yksinkertaisesti toimi ja esimerkiksi ajoneuvoissa johdotuksien pituus on useita metrejä. Demoamistarkoitukseen se on kuitenkin riittävän hyvä, mutta jos tuotetta on tarkoitus käyttää myös aidossa ympäristössä, jokin muu ratkaisu on löydyttävä. Tähän on useita ratkaisuja, joista projektin aikana eniten esille nousi SPI-väylän korvaaminen CAN-väylällä. Koska ajoneuvoista löytyy muutenkin CAN-väylä ja laitteiden lisääminen siihen on melko yksinkertaista, eikä segmentteihin perustuvien näyttöjen ohjaus vaadi suurta tiedonsiirtokapasiteettia, olisi se tähän tarkoitukseen ideaalinen. CAN-väylän soveltuvuus häiriöalttiiseen ympäristöön myös tukee sen valintaa.

Topologialtaan tällainen ratkaisu voisi olla samanlainen kuin tälläkin hetkellä, jossa yksi toimintalogiikan sisältävä laite ohjaa näyttöjä. Tässä tapauksessa ohjausviestit vain lähetettäisiin CAN-väylälle SPI-väylän sijaan. Toinen mahdollinen toimintatapa olisi, että jokaisen näyttölaitteen ohjainpiiri toimisi itsenäisesti CAN-väylää kuunnellen ja näytettävän tiedon sieltä poimien. Tilaa säästyisi yhden laitteen verran, eikä CAN-väylää kuormitettaisi näyttöjen ohjausviesteillä. Tällainen toimintatapa tosin vaatii eri tavalla toimivan ohjelmiston jokaiseen ohjainyksikköön riippuen minkälainen näyttö on kyseessä. Ohjelmisto voisi esimerkiksi olla samaan tapaan konfiguroitavissa kuin prototyypissä. Käytännössä muutokset ohjelmistoon olisi kuitenkin helpompi tehdä, jos ohjauksesta vastaisi vain yksi yksikkö. Sellaisen tiedon näyttämiseen, jota ei CAN-väylältä ole saatavissa, pitäisi joka tapauksessa käyttää ulkoista ohjausta.

CAN-väylän lisäksi jokin langaton siirtotapa, kuten Bluetooth tai WLAN saattaisi soveltua näyttöjen ohjaamiseen. Itsenäisten ohjausyksikköjen tapauksessa sitä voitaisiin käyttää esimerkiksi ainoastaan sellaisen tiedon välittämiseen, jota ei CAN-väylältä ole saatavissa. On myös olemassa adaptereita, joilla voidaan luoda BlueToothin avulla silta CAN-väylän datalle. Tätä voitaisi käyttää, jos näytön ohjauslaite pitäisi sijoittaa kauas CAN-väylästä. Langattoman siirtotavan soveltuvuutta tähän tarkoitukseen tulisi kuitenkin tarkemmin tutkia ennen sen käyttöönottoa.

Yllä esitetyt vaihtoehdot vaativat muutoksia Lumineqin ohjauspiireihin. Mikäli tämä ei ole mahdollista, voitaisiin vaihtoehtoisesti tehdä oma sovituskappale, joka välittäisi CAN-väylältä saadut ohjauskäskyt SPI-väylää pitkin Lumineqin ohjauspiirille. Jos kuitenkin SPI-yhteyttä halutaan välttämättä käyttää, niin viimeisenä vaihtoehtona voisi olla differentiaalisen tiedonsiirron käyttäminen kierretyssä parikaapelissa [16]. Tällä tavoin suurin osa häiriöistä saataisiin eliminoitua ja pituuden

ei pitäisi muodostua ongelmaksi. Huonona puolena tällaisessa vaihtoehdossa joutuisi johdotuksen tekemään erikseen jokaiselle näyttölaitteelle.

6.2.2 Ohjelmisto

Ohjelmistossa havaittiin muutamia kehittämiskohtia ja puutteita, jotka eivät kuitenkaan vaikuta merkittävästi sen toimintaan. Suurin osa niistä on melko triviaaleja ja helposti ratkaistavissa, mutta osassa vaaditaan jo melko suuria muutoksia ohjelmiston toimintaan.

Pääsäikeessä voitaisiin käyttää select-funktiota odotettaessa viestejä muilta säikeiltä sen sijaan, että käydään silmukassa pollaamassa eri viestijonoja. Näin prosessi jäisi automaattisesti odottamaan select-funktiossa mikäli mitään käsiteltävää ei ole. Tämä tosin vaatisi, että käytetään itse tehtyjen jonojen sijasta Linuxin tarjoamaa jonoa ja tiedostokuvainta, mikä näin jälkikäteen ajateltuna olisi saattanut olla järkevämpää. Muutoksen toteuttaminen nykyiseen ohjelmaan vaatisi jonkin verran aikaa, sekä testausta, mutta pidemmällä tähtäimellä se olisi todennäköisesti kannattavaa.

Kuten aliluvussa 5.1.3 todettiin, konfiguraatiotiedosto UI-kuuntelijan osalta sisältää paljon turhaa tietoa, joita käyttäjän ei pitäisi muokata. Oikeastaan lista käskyistä, niiden parametrien lukumäärä ja vastauksien muoto voitaisiin poistaa kokonaan ja asettaa ne kiinteiksi ohjelmakoodissa. Ainoastaan virheviestit ja niiden indeksit voisivat olla konfiguroitavissa. Yleisesti, vaikka YAML on ihmisystävällinen, konfiguraatiotiedostot saattavat olla melko työläitä käsitellä ja ylläpitää, joten tähän tarkoitukseen voisi kehittää graafisen työkalun ylläpitoa helpottamaan. Työkalu voisi olla jopa samassa ohjelmassa aliluvussa 5.4.1 mainitun TCP-testaustyökalun ohella.

Jos käyttöliittymä ei ole muodostanut TCP-yhteyttä, virheviestit CAN-väylältä ja näyttöyksiköstä kasautuvat jonoon. Ylivuotoa ei kuitenkaan pitäisi tapahtua, vaan jonon tullessa täyteen loput viestit tuhotaan, ja käyttöliittymän muodostaessa yhteyden jonossa olevat virheviestit lähetetään. Tästä seuraa, että jotkin virheviestit voivat jäädä huomaamatta, joskin se on melko epätodennäköistä. Kaikki virheviestit kuitenkin kirjataan lokitiedostoon, josta ne löytyvät myös ohjelmiston sammutuksen jälkeen. Jonossa olevien virheviestien lähetys voi aiheuttaa hämmennystä yhdistettäessä, joten yksi ratkaisu voisi olla sivuuttaa kaikki virheviestit silloin, kun yhteyttä ei ole muodostettu.

CAN-kuuntelijan konfiguraatiotiedostossa on määritelty tilat, joissa kutakin SPN-arvoa käytetään. Tämän listan perusteella kontrolliyksikkö päättää saapuneen CAN-viestin kohdalla tarvitseeko nykyisen tilan näytettäviä arvoja päivittää. Nykyisen

tilan löytyessä listalta kontrolliyksikkö laskee konfiguraatiotiedostossa olevan laskukaavan perusteella näytettävän arvon. Tilan näytettävään arvoon vaikuttavat SPN-arvot voitaisi kuitenkin selvittää laskukaavan perusteella, jolloin jokaisen SPN:n kohdalla ei tarvitsisi listata niitä tiloja, joissa sitä käytetään. Muutoksen voisi toteuttaa katsomalla konfiguraatiotiedoston lukuvaiheessa, mitkä SPN-arvot laskukaavassa mainitaan ja lisätä ne tilan tietoihin, josta niiden vertaaminen saapuviin viesteihin olisi mahdollista. Syy, miksi nykyiseen tapaan päädyttiin oli laskukaavan tarpeen ilmeneminen myöhemmässä vaiheessa. Alkuvaiheessa CAN-kuuntelija suodatti viestejä vertailemalla kontrolliyksikön nykyistä tilaa konfiguraatiotiedostossa SPN:n kohdalta löytyvään listaan tiloista. Suodatus kuitenkin siirrettiin pääsäikeelle, sillä vastaanotetut SPN-arvot haluttiin tallentaa riippumatta kontrolliyksikön tilasta. Joidenkin SPN-arvojen lähetystiheys CAN-väylällä ei ole riittävän tiheä, jotta tilaa muutettaessa käyttäjä ei huomaisi arvon hetkellistä puuttumista. Kun arvot tallennetaan tilasta riippumatta, voidaan tilaa vaihdettaessa hakea uuden tilan tuorein arvo välittömästi.

Näyttöjen käyttöliittymän logiikan sijainti on kyseenalainen mietittäessä näyttöjen geneeristä käyttötarkoitusta. Tällä hetkellä se sijaitsee kontrolliyksikössä, joka huolehtii navigoinnista ja valinnoista ikonien avulla. Demoamistarkoituksessa se toimii, mutta entäpä, jos johonkin käyttötarkoitukseen halutaan erilainen toimintalogiikka. Jatkon kannalta olisikin mielekkäämpää, mikäli toimintalogiikka tehtäisiin esimerkiksi erillisessä ohjelmassa ja kommunikointi tapahtuisi käyttäen käyttöliittymän TCP-protokollaa. Esimerkiksi käyttöliittymän ohjelmisto voitaisiin ajaa samalla laitteella kontrolliyksikön ja näyttöyksikön kanssa. Käyttöliittymän ohjelmisto voisi sisältää tällöin myös toimintalogiikan. Käyttöliittymän TCP-protokolla sallii näyttöjen täydellisen hallinnan, jolloin kontrolliyksikön tehtäväksi jäisi vain CAN-väylältä saatujen viestien käsittely ja mahdollinen välitys käyttöliittymälle tai näytöille. Tällä tavalla käyttöliittymän logiikan päivitys ei vaatisi muutoksia kontrolliyksikköön ja toimintalogiikan voisi tarpeen vaatiessa toteuttaa eri ohjelmointikielillä tai kokonaan eri alustalla.

Jatkon kannalta ohjelmiston suunnittelussa tulisi ottaa huomioon myös tietoturvallisuus. Tällä hetkellä mitään siihen liittyviä tarkastuksia ei tehdä ja ulkopuolisen on helppo päästä konfiguroimaan ohjelmistoa. Käyttöliittymän protokollakaan ei mitenkään tarkista, kuka komentoja lähettelee.

6.2.3 Rajapinnat

Ulkoiset rajapinnat mahdollistavat jo käytännössä kaiken tähän mennessä vaaditun toiminnallisuuden, ja niihin liittyvät kehitysideat ovat enemmän toimintojen laajen-

tamiseen keskittyviä. Laitteen käyttötarkoitus tuskin tulee rajoittumaan ainoastaan työkoneisiin ja ajoneuvotietojen näyttämiseen, joten jotta rajapintoja voisi käyttää myös muuhun tarkoitukseen, ne vaativat joitakin muutoksia. Seuraavaksi esitellyt jatkokehitysideat koskevat käyttöliittymän rajapintaa ja CAN-väylän rajapintaa.

Käyttöliittymän rajapinnassa käytetyssä protokollassa on joitakin puutteita, jotka pitäisi korjata. Parametrit erottavana välimerkkinä toimii tällä hetkellä kaksoispiste, mutta myös informaationäytössä voidaan näyttää kaksoispiste esimerkiksi kellonaikaa näytettäessä. Tämä on ratkaistu siten, että 7-segmenttinäyttöjä asettavan komennon saapuessa kaksoispistettä ei huomioida parametreja erottavana merkkinä. Järkevämpi tapa ratkaista tämä olisi käyttää jotain muuta välimerkkiä parametreille tai vaihtoehtoisesti eriskoismerkin käyttö, jolla ilmaistaan seuraavan merkin kuulumattomuus komentoon. Protokollassa käytetyt käskyn aloitus- ja lopetusmerkit voitaisiin korvata esimerkiksi jokaisen käskyn lopussa olevalla rivinvaihtomerkillä, jolloin käskyistä tulisi yksinkertaisempia käsitellä.

Käyttöliittymän rajapinta sallii tällä hetkellä vain yhden yhteyden kerrallaan. Tilanne, jossa näyttöä ohjaa useampi laite voisi olla mahdollinen, mikäli käyttöliittymä sisältäisi useita itsenäisesti toimivia laitteita. Usean yhteyden salliminen vaatii UI-kuuntelijaan melko suuria muutoksia. Yksi ratkaisu olisi, että jokaista yhteyttä kohden luotaisiin uusi säie, jolloin UI-kuuntelija pitäisi kirjaa yhteyksistä ja niiden säikeistä. Toinen vaihtoehto olisi luoda lista aktiivisista yhteyksistä, joita UI-kuuntelija kävisi läpi vuoron perään tai odottaisi viestiä select-funktiolla. Jälkimmäinen vaihtoehto voisi soveltua paremmin sulautettuun järjestelmään, sillä ensimmäisessä vaihtoehdossa jokainen uusi säie syö muistia ja resursseja.

CAN-rajapinta tukee tällä hetkellä ainoastaan J1939-standardin mukaisia viestejä. J1939 on yleisesti käytössä ainoastaan hyötyajoneuvoissa, kuten työkoneissa tai busseissa. Muiden protokollien käyttö vaatii muutoksia ohjelmistoon, mutta rakenteeltaan ohjelmisto voisi kuitenkin olla samankaltainen, jossa yksi säie lukee CAN-väylän tietoja. Joitakin J1939-protokollan tarjoamia tietoja ei automaattisesti lähetetä väylälle, vaan ne pitää erikseen pyytää. Tällaisia tietoja ovat esimerkiksi keskikulutus ja renkaiden ilmanpaine. Pyyntöjen lähettäminen ei ole erityisen monimutkainen prosessi, mutta laitteelle pitää silloin asettaa J1939-standardin mukainen osoite ja ohjelmistoon tehdä tarvittavat muutokset. J1939 tarjoaa mahdollisuuden dynaamiseen osoitteistukseen, mikä olisi staattista osoitetta järkevämpi vaihtoehto tällaiselle todennäköisesti jälkiasennettavalle näytölle.

CAN-väyliä voi myös olla monta samassa ajoneuvossa, eikä kaikkea tietoa aina ole saatavilla yhdellä väylällä. Usean CAN-väylän kuuntelu onnistuu, mutta se vaatii

muutoksia laitteistoon ja ohjelmistoon. Nykyinen kehitysalusta sisältää vain yhden CAN-väylän, joten se ei tähän käyttötarkoitukseen sovellu. Ohjelmiston rakenne sen sijaan mahdollistaa pienillä muutoksilla useamman CAN-väylän lukemisen. Ohjelmisto on myös melko pitkälle yhteensopiva muiden vastaavien Linux-pohjaisten kehitysalustojen kanssa, joten kehitysalustan vaihtaminen pitäisi sujua melko kivuttomasti.

6.2.4 Kehitysympäristö

Kehitysympäristö oli kokonaisuutena onnistunut ja ohjelmiston kehitys oli sujuvaa. Uudet ominaisuudet saatiin nopeasti käännettyä ja testattua. Kehityksessä käytetty Qt 4.8 oli kuitenkin käytettävyydeltään huono. Esimerkiksi ikkunoiden koon muuttaminen ja sulkeminen oli erittäin hankalaa. Uudempi versio olisi saattanut korjata ongelmat, mutta käytössä olleelle Ubuntun versiolle ei ollut valmiina saatavilla uudempaa versiota. Uudemman version olisi saanut itse kääntämällä, mutta tähän emme ajanpuutteen vuoksi ryhtyneet. Vaihtoehtoisia työkaluja tähän tarkoitukseen löytyy myös useita, joista nopean tarkastelun jälkeen kokeilun arvoisia voisivat olla Code::Blocks [8], Eclipse [35] tai emIDE [11].

Työkoneiden käyttöjärjestelmällä ei ollut juurikaan merkitystä, sillä käännöstyö tehtiin virtuaalikoneessa

6.2.5 Elektroniikka

Elektroniikan osalta jatkokehitys riippuu siitä, miten SPI-väylä korvataan. Kehitysalusta voisi joka tapauksessa olla pienempi, sillä nykyinen sisältää joitakin turhia ominaisuuksia, kuten monta USB-porttia, kaksi Ethernet-porttia, HDMI ym. Yksi vaihtoehto monista voisi olla BeagleBone Black [34], joka sisältää tarvittavat liitynnät ja on kokonsa puolesta varsin kompakti.

Lumineq-ohjainpiirin lattakaapeliliitin näytöille ja lattakaapeli yleisesti ei välttämättä ole paras vaihtoehto käytettäväksi ajoneuvoissa, jossa ne voivat joutua kovan rasituksen alle. Kaksirivinen 10-napainen liitin, johon tuodaan ohjainpiirin käyttöjännite. SPI-signaali vaatii myös käytännössä lattakaapelin käyttöä. Saatavilla olevien kaapeleiden johdinpaksuus on 28 AWG(engl. American Wire Gauge), mikä johtaa pidemmillä etäisyyksillä jännitehäviöön. Ratkaisuna tähän voisi olla käyttää sähkönsyöttöön erikseen paksumpia kaapeleita.

7. YHTEENVETO

Kaiken kaikkiaan prototyyppiä voidaan luonnehtia onnistuneeksi. Messuille tarkoitettu demo saatiin toimimaan niin kuin haluttiin, eikä suuria vastoinkäymisiä ollut. Projekti tuotti myös arvokasta tietoa jatkokehityksen kannalta. Tärkeimpänä mainittakoon ohjelmiston arkkitehtuuri, joka mahdollistaa helpon laajentamisen. Tämä osoittautui hyödylliseksi loppuvaiheessa, kun ilmaantui tarve kahdelle samanaikaiselle TCP-yhteydelle. Tämä saatiin nopeasti toteutettua luomalla toinen säie uudelle TCP-yhteydelle.

Ohjelmisto saatiin toteutettua rakenteeltaan sellaiseksi, että se voidaan konfiguroida toimimaan erilaisilla näytöillä. Helppo konfiguroitavuus oli tässä projektissa avainasemassa ja se toteutettiin käyttäen konfiguraatiotiedostoja, joilla määriteltiin näyttöjen sisältö ja CAN-väylältä kuunneltava tieto. Näytettävän tiedon kuuntelu onnistuu pienillä muutoksilla myös useammasta CAN-väylästä, joka voi olla tarpeen joissakin ajoneuvoissa.

Tuulilasissa sijaitsevat näytöt ovat ensimmäisiä laatuaan ja ovat projektin myötä osoittautuneet hyväksi ideaksi. Vaikka ne perustuvat kiinteisiin segmentteihin, eikä niitä voida käyttää yhtä monipuolisesti kuin pikseleihin perustuvaa näyttöä, ovat ne silti merkittävä askel kohti tulevia käyttökohteita. Tässä vaiheessa näyttöteknologia ei vielä ole kehittynyt riittävän pitkälle, mutta esimerkiksi Samsung on julkistanut joitakin läpinäkyviä OLED-näyttöjä [5], jotka voisivat tulevaisuudessa korvata nykyisen tekniikan. Tällaiseen näyttöön siirryttäessä tässä projektissa toteutetun ohjelmiston rakenne voisi pysyä samanlaisena, mutta näyttöjä ohjaava osuus pitäisi uusia.

Prototyyppi herätti Pilkingtonin potentiaalisten asiakkaiden kiinnostuksen jo ennen varsinaista esittelyä messuilla, josta johtuen Pilkington tilasi lisäerän prototyyppilaitteita. Ylimääräisiä prototyyppejä ryhdyttiin suunnittelemaan ja valmistamaan heinäkuun-syyskuun aikana.

LÄHTEET

[1] RS232 Tutorial on Data Interface and cables, ARC Electronics, 2010, verkkosivu. Saatavissa (viitattu 29.10.2015): http://www.arcelect.com/rs232.htm.

[2] O. Ben-Kiki, C. Evans, I. döt Net, YAML Ain't Markup Language Version 1.2, 2009, dokumentti. Saatavissa: http://yaml.org/spec/1.2/spec.pdf.

[3] Transparent Electroluminescent Displays, Beneq Products Oy, 2013, dokumentti. Saatavissa: http://lumineq.com/sites/default/files/documents/lumineq_tasel_brochure.pdf.

[4] Transparent electroluminescent Displays, Beneq Products Oy, verkkosivu. Saatavissa (viitattu 12.10.2015): http://lumineq.com/en/products/tasel.

[5] Business Wire, Samsung Display Introduces First Mirror and Transparent OLED Display Panels, 2015, artikkeli. Saatavissa (viitattu 15.9.2015): http://www.businesswire.com/news/home/20150609006775/en/Samsung-Display-Introduces-Mirror-Transparent-OLED-Display.

[6] S. K. Card, G. G. Roberston, J. D. Mackinlay, The information visualizer: An information workspace, *Proceedings of the SIGCHI Conference on Human Factors in Computin Systems*, 1991, pp. 181–186.

[7] G. Cena, A. Valenzano, Protocols and Services in Controller Area Networks, CRC Press, 2014.

[8] Code::Blocks, verkkosivu. Saatavissa (viitattu 15.9.2015): http://www.codeblocks.org/.

[9] Jack B. Copeland, The Church-Turing Thesis, The Stanford Encyclopedia of Philosophy (Summer 2015 Edition). Saatavissa: http://plato.stanford.edu/entries/church-turing/.

[10] Linux kernel with J1939 support, EIA Electronics, verkkosivu. Saatavissa (viitattu: 6.1.2015): https://github.com/kurt-vd/linux.

[11] emIDE, verkkosivu. Saatavilla (viitattu 15.9.2015): http://www.emide.org/.

[12] YAML 1.2, Clark C. Evans, verkkosivu. Saatavissa (viitattu 15.9.2015): http://yaml.org/.

[13] Dimitri van Heesch, Doxygen, verkkosivu. Saatavissa (viitattu 15.9.2015): http://www.stack.nl/~dimitri/doxygen/index.html.

[14] IEEE 802.3 Ethernet, IEEE Standards Association, verkkosivu. Saatavissa (viitattu 29.10.2015): http://standards.ieee.org/about/get/802/802.3.html.

[15] Michael Kerrisk, pthreads, Linux Programmer's Manual, verkkosivu. Saatavissa (viitattu 2.7.2015): http://man7.org/linux/man-pages/man7/pthreads.7.html.

[16] Thomas Kugelstadt, Extending the SPI bus for Long-Distance Communication, Texas Instruments Incorporated, 2011, dokumentti. Saatavissa: http://www.ti.com/lit/an/slyt441/slyt441.pdf.

[17] J1939 introduction, Kvaser, verkkosivu. Saatavissa (viitattu 13.4.2015): http://www.kvaser.com/about-can/higher-layer-protocols/j1939-introduction/.

[18] Leap Motion, Leap Motion Inc., verkkosivu. Saatavissa (viitattu 12.10.2015): https://www.leapmotion.com/.

[19] Carrier Sense Multiple Access Collision Detect (CSMA/CD) Explained, Learn Networking, 2008, verkkosivu. Saatavissa (viitattu 29.10.2015): http://learn-networking.com/network-design/carrier-sense-multiple-access-collision-detect-csmacd-explained.

[20] Shunting yard algorithm (C), LiteratePrograms, 2013, verkkosivus. Saatavissa (viitattu 15.9.2015): http://en.literateprograms.org/Shunting_yard_algorithm_%28C%29.

[21] NMEA 2000 Standard, National Marine Electronics Association, verkkosivu. Saatavissa: (viitattu 13.4.2015): http://www.nmea.org/content/nmea_standards/nmea_2000_ed3_10.asp.

[22] MilCAN Standard, MilCAN, verkkosivu. Saatavissa: (viitattu 13.4.2015): http://www.milcan.org/.

[23] MYIR MYD-AM335X Development Board, MYIR Tech Limited, verkkosivu. Saatavissa (viitattu: 15.9.2015): http://www.myirtech.com/list.asp?id=466.

[24] The Shunting Yard Algorithm, P. Oser, verkkosivu. Saatavissa (viitattu 14.10.2015): http://www.oxfordmathcenter.com/drupal7/node/628.

[25] Rcvfrom, Linux Man Page, verkkosivu. Saatavissa (viitattu: 15.9.2015): http://linux.die.net/man/2/recvfrom.

[26] Embest DevKit8600 Development Board, Shenzhen Embest Technology Co., Ltd., verkkosivu. Saatavissa (viitattu: 15.9.2015): http://www.embest-tech.com/product/evaluation-boards/devkit8600-evaluation-board.html.

[27] LibYAML, Kirill Simonov, verkkosivu. Saatavissa (viitattu 15.9.2015): http://pyyaml.org/wiki/LibYAML.

[28] J1939/21 Data Link Layer, Society of Automotive Engineers, verkkosivu. Saatavissa (viitattu: 10.6.2015): http://standards.sae.org/j1939/21_201012/.

[29] J1939/71 Vehicle Application Layer, Society of Automotive Engineers, verkkosivu. Saatavissa (viitattu: 10.6.2015): http://standards.sae.org/j1939/71_201506/.

[30] J1939/73 Application Layer - Diagnostics, Society of Automotive Engineers, verkkosivu. Saatavissa (viitattu: 10.6.2015): http://standards.sae.org/j1939/73_201307/.

[31] The SAE J1939 Communications Network, Society of Automotive Engineers, verkkosivu. Saatavissa (viitattu 1.9.2015): http://www.sae.org/misc/pdfs/J1939.pdf.

[32] Serial Peripheral Interface (SPI), Sparkfun, verkkosivu. Saatavissa (viitattu 22.10.2015): https://learn.sparkfun.com/tutorials/serial-peripheral-interface-spi.

[33] sscanf, cplusplus.com, verkkosivu. Saatavissa (viitattu: 15.9.2015): http://www.cplusplus.com/reference/cstdio/sscanf/.

[34] BeagleBone Black, The BeagleBoard.org Foundation, verkkosivu. Saatavissa (viitattu 15.9.2015): http://beagleboard.org/black.

[35] Eclipse, The Eclipse Foundation, verkkosivu. Saatavilla (viitattu 15.9.2015): https://eclipse.org/.

[36] Iproute2, The Linux Foundation, verkkosivu. Saatavissa (viitattu 15.9.2015): http://www.linuxfoundation.org/collaborate/workgroups/networking/iproute2.

[37] Spidev, The Linux Kernel Archives, dokumentti. Saatavissa (viitattu 29.10.2015): https://www.kernel.org/doc/Documentation/spi/spidev.

[38] Qt The IDE, The Qt Company, verkkosivu. Saatavissa (viitattu 13.10.2015): http://www.qt.io/ide/.

[39] CANalyzer, Vector Informatik GmbH, verkkosivu. Saatavissa (viitattu 15.9.2015): http://vector.com/vi_canalyzer_en.html.

[40] Wilfried Voss, A Comprehensible Guide to J1939, Copperhill Technologies Corporation, 2008. 113 s.

LIITE 1. YAML-KONFIGURAATIO

YAML GUIDE

Project Glass

Miikka Juomoja & Tommi Rekosuo
Tampere University of Technology
Version 1.0

Revision history

Date	Version	Author	Comments
2.6.2015	1.0	MJ	Initial documentation based on YAML documentation txt-file.
4.6.2015	1.1	TR	Added communication.yaml and can.yaml field descriptions

Table of Contents

Revision history....2
1.About this document....4
2.YAML usage tutorial....5
2.1.YAML writing tutorial....5
a)# is one line comment....5
b)Pages....5
c)Key – Value pair....6
d)Lists....6
e)Objects....6
f)Other things to be mentioned....7
3.Filling the data....8
3.1.Data fields in control.yaml....8
3.2.Data fields in communication.yaml....9
3.3.Data fields in can.yaml....9

1. About this document

This document describes how to build a custom YAML configuration file to allow the system to work properly. The YAML configuration file holds additional information about the individual lamps. The additional systems allow lamps to be grouped, iconized or marked as a digit groups.

The YAML is easy-to-use and easy-to-understand script language. The point of the system is to use same codebase throughout the whole system and if there are changes in the product e.g. Different glass combinations, then the systems configuration files are the only ones which need to be updated. The configuration files are built to be simple as possible so even people with no programming background can update the files with ease.

YAML can be edited with almost all text processing software e.g. Notepad, but the recomendation is Notepad++ on Windows or Gedit/Nano in Linux. Gedit/Nano comes with all Linux operation systems by default but the Notepad++ needs to be downloaded from the Notepad++ website. All of the softwares are free to use and open source.

2. YAML usage tutorial

2.1. YAML writing tutorial

There are some aspects about how to write the YAML configurations. This tutorial shows all the different commands and types which is crucial to know before starting to edit the YAML.

Setting up the files are simple. The path is always the same as the program's path (Middleware CAN or SPI) and in the same folder there has to be folder called config. In the config forlder there has to be the following configuration files:

- control.yaml
- can.yaml
- communication.yaml

a) # is one line comment

First thing in the documents are comments. Commenting is crucial in making YAML code. The files get rather large so additional information needs to be passed from editor to editor. Sometimes there are parts which are best to be ”commented out” than just deleting them because some parts can be used in different instance. When the line starts with hashtag (#) the line is not used in the read process so it is only for people who edit the file.

Example:

This is a comment

b) Pages

The pages are a way to sort data into a specified chunks. Page starts with ”---” and ends with ”...” The system reads the information by the pages so the correct data must be inside correct page. Projects style indicates that after ”---” there is a comment of pages type on the next line.

Example:

This is a page

...

The different page types are:

- Control.yaml
  - Settings = Global settings which affect certain aspects of the software

    - Lamps = Information about screens lamps
    - Icons = Set of lamps which are used in navigation mode
    - Groups = Set of lamps which are not used in navigation but which need to be light simultaniously
    - Digits = Listing of groups which are part of a digit group. Digit groups automatically display numeric values
- Communication.yaml
    - Communication information has the commands and actions of the Ethernet connection
- Can.yaml
    - CAN reading page tells the system which can messages are read from the CAN
    - State page which has the information of each mode and how to calculate the value and in which format it is displayed.

c) Key – Value pair

The data is stored in key value pairs which are separated by double dot (”:”). On the left side there is a key which indicates the data type. The value is on the right side and the format is determined by the type. The data type can be numeric, string, list (uses square brackets ”[” and ”]”) or objects (start with ”-”). More detailed information about the data is presented at chapter 3.

Example:

dtype: 7

d) Lists

Lists start and end with square brackets. They hold information which are separated by comma. The data can be the same things as a value can be. In applications case the only usage is to list unique names (more about that on later chapters).

Example:

glamps: [Temp Lamp, Engine Middle]

e) Objects

Objects start with a line at the start of the first key – value pair. The idea is to make a ”shopping list” but in this case the shopping list contains product name, size, weight and price for each item listed. Therefore the similiar objects which are categorized to

contain the same amount and name of keys, in key – value pair, are objects.

Example:

```
- lname:      GPS Icon        # 1.st object
  scrid:      1
  lmpid:      40

- lname:      GPS Circle      # 2nd. object
  scrid:      1
  lmpid:      39
```

f) Other things to be mentioned

There are a set of rules which has to be follower. These rules affect the system and will break the system.

1) All the keys are case sensitive. Like in all programming languages the case sensitivy must be noted.

2) Tabs are not allowed. When using tabulator, make sure the your tabs button is formatted to print x amount of spaces.

3) Best way to start is to copy. Copy and paste objects and just fill them

4) Newline is a divider between key – value pairs so one item must be on a new line

3. Filling the data

3.1. Data fields in control.yaml

Data fields in control.yaml are grouped by pages. Each page has a set of similar objects. Note that the **Unique names** must be unique so there should be only one with certain name.

- Settings page
  - maxvoltage: Maximum value of voltage dimming
  - maxframerate: Maximum value of framerate dimming
  - selectflick: Flicktime (time when the lamp is on or off) of navigation process. 100 = 1 second
  - Level values: The priority of the type. If priority is lower than other types priority, then the larger priority wins. So if a group is on and the priority is larget than lamp, then the lamp command is not affecting the screens.
- Lamp page
  - lname: **Unique name** of the lamp.
  - scrid: The screen number where the lamp is located
  - lmpid: The lamp's pin number at the screen (documentation)
- Icons page
  - iname: **Unique name** of the icon
  - ilamps: List of lamps which form the icon (list items are lname values)
- Groups page
  - gname: **Unique name** of the group
  - glamps: List of lamps which form the group (list items are lname values)
- Digits page
  - dgname: **Unique name** of the digit group
  - digits: List of single digit items
  - dtype: Type of the digit item (7 = 7-sequence display, single = dot/double dot and zero = single lamp which forms zero)
  - gdigits: Unique name of a group (7) or lamp (single, zero) which holds the information about the lamps

3.2. Data fields in communication.yaml

Communication.yaml lists all the possible commands for the tcp interface and related information. It also lists possible error names that are emitted if there's a diagnostics message received from the can bus. The only thing that should be edited is this list of error names. The error names have to be the same as the ones in can.yaml under diagnostics section.

- Main page
  - commands: List of commands
  - command: String form of a command
  - id: Internal id of the command (hardcoded, **do not change!**)
  - parts: How many parts does the command have separated by colon (don't change)
  - reply: Ordered list of what to include in the reply. Values can be:
    - cur_pos: current cursor position
    - cur_scr: current screen where cursor is
    - id: ID of the lamp/group/screen/icon related to the command
    - value: value of the lamp/group/screen/icon related to the command
    - state: current state which is the icon index of the lower right screen (for example RPM or Oil icon)
  - errors: List of error names. This list is used to index the error message when sending diagnostics messages received from the can bus.
  - error: Error name (should be the same as used in can.yaml diagnostics section

3.3. Data fields in can.yaml

Can.yaml is used to configure which J1939 SPNs to listen from the CAN bus. It also has a diagnostics list of what should happen if a certain spn appears in a DM1 diagnostics message. Then there's the list of different modes and what information to display on each one. Each mode is linked to one icon.

- CAN reading page
  - pgn: List of pgns to accept. Each pgn holds a list of spns which are used.
    - spn: Number of the spn
    - offset: Offset of the spn from the start of pgn packet
    - length: Length of spn in bytes
    - mode: List of different modes (icons) where this mode is used. In other words if this spn changes, which mode needs to recalculate it's display value.

- diagnostics: This is a list of spns that we are interested if they appear in a diagnostics message.
  - spn: The spn number
  - valuemin: Optional minimum value for the raw spn value
  - valuemax: Optional maximum value for the raw spn value
  - lgroup: Lamp group which starts blinking when diagnostics message is received or when the raw value goes below minimum or over maximum
  - error: Error name that matches one from communication.yaml. The index of this name is sent through the tcp interface.

- State page
  - startmode: The starting state/mode which is an icon name
  - startcursor: The starting cursor position which is also an icon name
  - modes: list of different modes/states
    - mode: Icon name corresponding the mode
    - equation: How the displayed value is calculated. You can use received spn values by using $-sign followed by the spn number. For example value of spn 120 is ”$120”. There's also support for moving average value, ”m120” and total average value from current runtime ”a120”. Currently only basic calculations are supported (+, -, *, /,%,^). Negative numbers are marked with a '_'-character before the number (for example -10 is _10).
    - dgname: What digit display to use when showing the value
    - format: What format to use when displaying the value. For example showing four most significant numbers would look like ”nn nn”. There's a space because there could be a colon between the numbers. ”nn:nn” could be used to show time.
      - n: number
      - space: empty
      - :: Display colon.

LIITE 2. KÄYTTÖLIITTYMÄN PROTOKOLLA

TCP Interface Documentation

Project Glass

Tommi Rekosuo
Tampere University of Technology
Version 1.0

Revision history

Date	Version	Author	Comments
1.6.2015	1.0	TR	Initial documentation based on interface description txt-file.

Table of Contents

Revision history....2
1.About this document....4
2.Communication....4
2.1.General....4
2.2. Packets....4
3.Packet format....4
3.1. General packet format....4
3.2. Sending commands....4
3.3. Error info....4
3.4. ACK....5
3.5. NACK....5
4.Commands....5
4.1. List of commands....5
4.2. Requesting info....7
4.3. Current state....7
5. Error codes....8
6.Examples....8

1. About this document

This document describes the TCP interface for controlling the glass. The interface is used by the UI to change state of the device, but it can also be used to control individual lamps and the state machine can be implemented somewhere else if it does not function as desired.

2. Communication

2.1. General

Packets are transmitted over a TCP connection. The middleware software listens on port 6542(default) and accepts one connection at a time. This connection is used to transmit packets both ways according to rules in sections 2.2 and chapter 3. The default listening port can be changed in configuration file communication.yaml.

2.2. Packets

The server sends a response for each command sent. If the command was successful an ACK-packet(Section 3.5) is sent. If the command is somehow incorrect a NACK-packet(Section 3.6) is sent.

2.3. Info on Development board

The Ethernet port used on MYIR Development board is eth0, but there's a labeling error on some boards (at least in revision 2014200) where the eth0 is actually eth1 and eth1 is eth0. Physically it's the second one from power plug.

3. Packet format

Section 2.1 describes the general packet format and sections 3.1-3.3 describe the data portion of the packet. Section 3.5 describes the ACK-packet and section 3.6 describes the NACK-packet.

3.1. General packet format

1 byte	N bytes	1 byte
Packet start	Data portion(command + additional data)	Packet end

Packet start: An exclamation mark (!, ASCII 33) is used as indication of packet start.

Data portion: Consists of a command which is a string(see section 3) and additional data depending on command.

Packet end: A semicolon (;, ASCII 59) is used as indication of packet end.

3.2. Sending commands

Commands are sent independently and there are no handshakes or other necessities to do before commands can be sent. Just open a TCP-connection to the device and send the commands. The device always responds with an ACK or NACK. Portions of data are separated by colon. For example if lamp number 1 should be set on command !SetLamp:1:255; is sent.

3.3. Error info

This packet is used if a diagnostic code has become active or deactivated from CAN-bus (J1939 DM1). The packet is of following format:

1 byte	N bytes	1 byte
Packet start	Error:Error code:Status	Packet end

Error code: A number which identifies the error. See section 5.

Status: A number which describes whether the error is active or not. See section 5.

3.4. ACK

ACK is sent if the packet was received successfully and the command is valid. The ACK is of following format:

1 byte	N bytes	1 byte
Packet start	Ack:Received command[: optional data]	Packet end

Received command: The command which this Ack is a response to
Optional data: Optional data depends on the command. This can include cursor position, values that were set by the command and so on.

3.5. NACK

NACK is sent if received command is not in proper format or something else failed while processing the command. NACK uses following format

1 byte	N bytes	1 byte
Packet start	Nack:Received command[: optional data]	Packet end

4. Commands

4.1. List of commands

Sending commands should be in this format:

!command[:optional:optional];

The part in square brackets might be used on some commands (excluding the brackets of course).
The Ids are dynamically assigned based on their position in yaml files. To get a list of Ids see section 4.2.
If you need to set a certain lamp according to Lumineq documentation you can use the DebugLamp command where you specify the lamp number and which screen it is on.

Command	Additional Data	Reply	Description
Next	-	Ack with cursor position	Used for traversing icons
Previous	-	Ack with cursor position	Used for traversing icons
Select	-	Ack with selected icon	Used for selecting icon
NextScreen	-	Ack with current screen	Used for traversing screens. Lights up the first icon of the screen.
PreviousScreen	-	Ack with current screen	Used for traversing screens. Lights up the first icon of the screen.
SetScreen	ID:ON/OFF/FLICK	Normal Ack	Used for setting screen. ID is the id of the screen. ON/OFF/FLICK is a value between 0 and 255. 0 means off and 255 means constantly on. Any value between starts blinking the lamp . The rate of blinking is determined on the value which is in 1/100 of second.
SetIcon	ID:ON/OFF/FLICK	Normal Ack	Used for setting icon. See SetScreen description.
SetGroup	ID:ON/OFF/FLICK	Normal Ack	Used for setting group. See SetScreen description.
SetLamp	ID:ON/OFF/FLICK	Normal Ack	Used for setting lamp. See SetScreen description.
SetDigits	ID:Value	Normal Ack	Set value to digits. The value is in the following format: 12:34 Space means that the digit is not in use so for example the number 5432 would be ”54 32”. Letters can be used if it's possible to show them on a 7-segment display.
RequestInfo	Lamps/Groups/Icons / Digits	Returns list of ids and their names	0 = Lamps 1 = Groups 2 = Icons 3 = Animations 4 = Screens 5 = Digit displays 6 = Error codes
SetScrBright	ID:Value	Normal Ack	Used for setting individual screen brightness. Uses voltage dimming and the range is from 0 to 255.
SetScrBrightHz	ID:Value	Normal Ack	Used for setting individual screen brightness. Uses frame rate dimming and the range is from 0 to 2047

CurrentState	-	Cursor position and selected icon in lower right screen.	Used for retrieving the current state. Returns current cursor position and selected icon.
UnlightAll	-	Normal Ack	Closes all lamps.
NavigateTo	ID	Normal Ack	Navigates to and selects an icon.
DebugLamp	ID:SCRID:VALUE	Normal Ack	Used for setting a specific lamp on some screen
SetSpn	SPN:VALUE	Normal Ack	Used for setting spn through tcp interface as it would appear to come from can. This is for demonstration purposes only!.

4.2. Requesting info

When requesting lamp/group/icon/screen/digits/error info the reply is in the following format:

1 byte Packet start	N bytes Ack:RequestInfo:ID:name,ID:name,...	1 byte Packet end

ID is the number that is used to identify the lamp/group... It is dynamically assigned when starting program according to the order of things in yaml, so this information can be different from previous run. Name is the name of the lamp/group... and it is defined in the yaml configuration.

4.3. Current state

Current state is sent when requesting it by command !CurrentState; or when connection to the server is established it sends it automatically.

The reply is in the following format:

!CurrentState:cursorpos:state;

where state is the currently selected icon id (only lower right info screen icon selections are stored) and cursorpos is the current position of the cursor.

5. Error codes

Error codes are generally only related to diagnostic messages received from can bus. Each error message looks like this:

!Error:error code:status;

The error code is a number which corresponds to a name of the error. The status is either 0 = not active and 1 = active.

The corresponding names can be requested with:

!RequestInfo:6;

which returns a list described in section 4.2.

The names are configured in communication.yaml and these names are then used in can.yaml to name each error message. The error codes are assigned runtime so they can change between different runs if communication.yaml is changed (ID is determined from order of the names, same as with lamp, group... and other ids).

6. Examples

Here are some example commands:

Setting a lamp with id 1 on:

```
!SetLamp:1:255;
```

Setting group with id 5 blinking at 1 sec interval:

```
!SetGroup:5:100;
```

Setting screen 3 off:

```
!SetScreen:3:0;
```

Traversing to next icon and selecting it:

```
!Next;
!Select;
```

Setting digit display with id 2 to display ”12:30”

```
!SetDigits:2:12:30;
```

Setting digit display with id 2 to display ”123”

```
!SetDigits:2:12 3 ;
```

Setting lamp number 2 on screen 3 on:

```
!DebugLamp:2:3:255;
```

LIITE 3. NÄYTTÖYKSIKÖN RAJAPINTA

```
0-------------------------------0
|      OVERALL INFORMATION      |
0-------------------------------0
```

This document is about the internal messaging system inside the two Middleware processes.

VERSION	EDITOR	DATE	CHANGES
0.1	MJ	5.1.15	The first version of the documentation.
0.2	MJ	16.1.15	Corrected the message data, added voltage dimming and removed dimming at lamp control
0.3	MJ	21.1.15	Added Screen control, lamp/ group flicking, digit data and unglighting segments
0.3	TR	6.2.15	Added navigation mode command, added details of each command
0.4	MJ	12.2.15	Edited digit command and icon command
0.5	MJ	2.3.15	Removed Unlight all cmd (do with it screen cmd)
0.6	MJ	8.5.15	Added return message and added success
1.0	MJ	1.6.15	Removed animation and added it to free for use

Table of Contents:

1.1 OVERVIEW
2.1 COMMAND DETAILS
3.1 MODE NUMBERING
4.1 RETURN MESSAGES
5.1 OTHER VARIANCES

```
0-------------------------------------------------------------------------------0
```

```
0-------------------------------0
|         1.1 OVERVIEW          |
0-------------------------------0
```

The messaging system base is presented here. All the messages are in little-endian format.

```
|Command | ID [2 bytes]     | Succ   | Info
┌-------------------------------------------┐
| 1 Byte |        |        | 1 Byte  | 1 Byte |
└___________________________________________┘
```

Time = Time measured in scale of 0.1 seconds.

Command = Command byte which tells the program what to do:

- 0x00 = Free for use
- 0x01 = Screen control
- 0x02 = Icon control
- 0x03 = Lamp control
- 0x04 = Group control
- 0x05 = Digit

```
                        - 0x06  =       Set maximum dimming (Hz)
                        - 0x07  =       Set maximum dimming (Voltage)
                        - 0x08  =       Navigation mode
                        - 0x09  =       Return message (ONLY SPI --> CAN)
                        - 0x0A  =       Debug light

ID              = Informational data:
                        - 0x00  =       Free for use
                        - 0x01  =       Screen ID (CS)
                        - 0x02  =       Icon ID
                        - 0x03  =       Lamp ID
                        - 0x04  =       Group ID
                        - 0x05  =       Data
                        - 0x06  =       Framerate dimming value
                        - 0x07  =       Voltage dimming value
                        - 0x08  =       ID
                        - 0x09  =       Type of error/ Type of return
                        - 0x0A  =       HIGH BYTE = Screen, LOW lmp ID on screen

Success         = if 1, then the successfull processing needs to return ACK
                  (THIS IS ONLY FOR MESSAGES FROM CAN --> SPI)

Info            = Additional information:
                        - 0x00  =       Free for use
                        - 0x01  =       0 = kill, 255 = start, 1-254 flick 0.1s
                        - 0x02  =       0 = kill, 255 = start, 1-254 flick 0.1s
                        - 0x03  =       0 = kill, 255 = start, 1-254 flick 0.1s
                        - 0x04  =       0 = kill, 255 = start, 1-254 flick 0.1s
                        - 0x05  =       Digit ID
                        - 0x06  =       Screen ID (CS)
                        - 0x07  =       Screen ID (CS)
                        - 0x08  =       Type 7 bits | on/off 1 bit
                        - 0x09  =       Data numbering (See below)
                        - 0x0A  =       Value



0-------------------------------------------------------------------------------0

0---------------------------------0
|        2.1 COMMAND DETAILS      |
0---------------------------------0


Command                     ID                      Info
-------------------------------------------------------------------------------
0x00 Free for use
0x01 Screen control         Screen ID               0 off, 255 on,
                                                        1-254 flick rate in 0.1s

0x02 Group control          Group ID                0 off, 255 on,
                                                        1-254 flick rate in 0.1s

0x03 Lamp control           Lamp ID                 0 off, 255 on,
                                                        1-254 flick rate in 0.1s

0x04 Digit                  1st bit ddot1           Digit ID
                            2nd bit ddot2
                            Last 14 bits = value

0x05 Set Maximum dimming    Dimming value           Screen ID (CS)
0x06 Set Maximum dimming    Dimming value           Screen ID (CS)
0x07 Unlight all segments   Screen ID (0 = all)     NOT USED

0x08 Navigation mode        ID                      Type 7 bits
                                                        0 Icon
                                                        1 Screen
                                                    On/off 1 bit
                                                        0 Off
                                                        1 On

0x09 Return                 Type (see below)        Dynamic data (see below)


0-------------------------------------------------------------------------------0

0-------------------------------0
```

4.1 RETURN MESSAGES

These messages are the return messages from the SPI to CAN so the CAN code can return information into the UI.

CMD = Always the same (see before)
ERR = Return message type (see below)

Return message types:

0. SUCCESS/ UNSUCCESS
1. NAVIGATION MESSAGE (current screen and icon)
2. FIFO FAILURE (from CAN to SPI)
3. NO ACCESS (priority is lower than the reserved)
4. DRIVER SEND FAILED
5. ITEM NOT FOUND (Basicly YAML error)
6 YAML ERROR

0 SUCCESS/ UNSUCCESS:

CMD	ERR	ISSUCC	TYPE	ID	DATA
1 Byte	[2 bytes]	1 Byte	[2 bytes]	[2 bytes]	[2 bytes]

ISSUCC = 0 if successfull
1 if failure
TYPE = Ticket type (lamp, group, icon, ...)
ID = Ticket ID
DATA = Ticket's data value

1 NAVIGATION MESSAGE:

CMD	ERR	SCREEN	ICON
1 Byte	[2 bytes]	[2 bytes]	[2 bytes]

SCREEN = In which screen the navigation is currently (ID)
ICON = In which icon the navigation is currentyl (ID)

2 FIFO FAILURE:

CMD	ERR
1 Byte	[2 bytes]

3 NO ACCESS:

CMD	ERR	TYPE	ID	TYPE	ID
1 Byte	[2 bytes]	[2 bytes]	[2 bytes]	[2 bytes]	[2 bytes]

TYPE 1 = Ticket type (lamp, group, icon, ...) from the CAN
ID 1 = Ticket ID from the CAN
TYPE 2 = Ticket type (lamp, group, icon, ...) which has the reserve
ID 2 = Ticket ID which has the larger priority reserve

4 DRIVER SEND FAILED:

CMD	ERR
1 Byte	[2 bytes]

5 ITEM NOT FOUND:

CMD	ERR	TYPE	ID	TYPE	ID
1 Byte	[2 bytes]	[2 bytes]	[2 bytes]	[2 bytes]	[2 bytes]

TYPE 1 = Ticket type (lamp, group, icon, ...) from the CAN
ID 1 = Ticket ID from the CAN
TYPE 2 = Ticket type (lamp, group, icon, ...) which was not found
ID 2 = Ticket ID which was not found

6 YAML ERROR:

CMD	ERR	TYPE	ROW	MESSAGE	END
1 Byte	[2 bytes]	[1 Byte]	[2 bytes]	[30 bytes]	[1 byte]

```
        TYPE    = YAML error type:
                          0: MEMORY ERROR (Calloc failure)
                          1: TWO SAME NAMES
                          2: WEIRD DATA INPUT (not ASCII)
                          3: KEY TYPO
                          4: ITEM NOT FOUND (65535)
                          5: TEXT OUT OF BOUNDS
                          6: NUMERIC DATA OUT OF BOUNDS
                          7: PARSE ERROR
        ROW     = Error row number
        MESSAGE = The data in the line
        END     = Reserved for the '\0' character
```

0--0

0------------------------------0
| 5.1 OTHER VARIANCES |
0------------------------------0

This chapter is about the all the variances from the base, what has changed and where it is used.

..::Ticket::..

TIME	CMD	ID	SUCC	INFO
[4 Bytes]	1 Byte	[2 bytes]	1 Byte	1 Byte

In ticket system there are two major changes. First the messages are changed into structs. The second change is that the time value is added to the start.

0--0