Osaamista
ja oivallusta
tulevaisuuden
tekemiseen

Topi Orpana

Adaptiivisen vakionopeudensäätimen suunnittelu ja rakentaminen

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Ajoneuvotekniikka

Insinöörityö

2.3.2020

Tekijä Otsikko Sivumäärä Aika	Topi Orpana Adaptiivisen vakionopeudensäätimen suunnittelu ja rakentaminen 28 sivua + 2 liitettä 2.3.2020
Tutkinto	Insinööri (AMK)
Tutkinto-ohjelma	Ajoneuvotekniikka
Ammatillinen pääaine	Autosähkötekniikka
Ohjaaja	lehtori Vesa Linja-aho

Tässä opinnäytetyössä rakennettiin adaptiivinen vakionopeudensäädin ajoneuvoon, jossa on valmiina perinteinen vakionopeudensäädin. Ajoneuvona oli käytössä vm. 2003 Audi A6. Välineinä opinnäytetyössä käytettiin välineitä, jotka ovat helposti kaikkien saatavilla.

Työssä perehdytään ajoneuvon väylätekniikkaan, etäisyysmittareihin, sulautettuihin järjestelmiin sekä hieman 3D-tulostukseen.

Pääosassa työssä on yhden piirilevyn minitietokone, Raspberry Pi 4 B, jota käytettiin etäisyysmittarin mittaustulosten vastaanottamiseen, ajoneuvon CAN-väylän lukemiseen ja ajoneuvon oman vakionopeudensäätimen ohjaukseen.

Raspberry Pin kosketusnäytön telineen valmistukseen käytettiin työssä 3D-tulostusta. Telineen suunnittelussa hyödynnettiin navigaattorin imukuppitelinettä, johon mallinnettiin sopiva välikappale, joka kiinnittyi näytön VESA-standardin mukaisiin kiinnityskohtiin.

Työssä käydään läpi koko kehitysprosessi ajoneuvoväylän etsimisestä valmiin tuotteen testaukseen. Turvallisuussyistä adaptiivista vakionopeudensäädintä ei kytketty auton jarrujärjestelmään lainkaan, koska haluttiin välttää tahattomat äkkijarrutukset, joista olisi saattanut olla vaaraa liikenneturvallisuudelle.

Yhteenvetona todetaan, että adaptiivisen vakionopeudensäätimen rakennus itse on mahdollista melko edullisestikin, joskin tietyin rajoituksin.

Avainsanat	CAN-väylä, sulautetut järjestelmät, LiDAR, 3D-tulostus, Python-ohjelmointi

Author	Topi Orpana
Title	Designing and Building an Adaptive Cruise Control
Number of Pages	28 pages + 2 appendices
Date	2 March 2020
Degree	Bachelor of Engineering
Degree Programme	Automotive Engineering
Professional Major	Automotive Electrics
Instructor	Vesa Linja-aho, Senior Lecturer

The aim of this thesis was to build an adaptive cruise control for a vehicle that is fitted with a traditional cruise control using equipment that is easily available for anyone.

This thesis deals with automotive bus technology, distance meters, embedded systems and 3D printing. The thesis explains the whole development process from finding the CAN bus in the vehicle to testing the ready product. The main focus was on the minicomputer Raspberry Pi 4 B that was used to receive measurements from the distance meter, read messages on the vehicle´s CAN bus and operate the existing cruise control in the vehicle. For security reasons the adaptive cruise control was not connected to the brake system of the vehicle to prevent unwanted sudden stops.

3D printing was used in manufacturing a stand for the Raspberry Pi touch screen. A navigator suction cup mount was fastened to the windshield of the vehicle and an adapter was designed and 3D printed to secure the screen to the mount.

In conclusion, building an adaptive cruise control is possible at a relatively low cost, although with certain limitations, such as the radar not functioning properly in low temperatures or losing the vehicle ahead because of the narrow field of view of the radar.

Keywords	CAN bus, embedded systems, LiDAR, 3D printing, Python programming

Sisällys

Lyhenteet

1 Johdanto 1

2 Vakionopeudensäätimen historia 2

2.1 Ensimmäiset vakionopeudensäätimet 2
2.2 Vakionopeudensäätimen yleistyminen 2
2.3 Adaptiivinen vakionopeudensäädin 2

3 Vakionopeudensäätimen toiminta 3

3.1 Perinteinen vakionopeudensäädin 3
3.2 Adaptiivinen vakionopeudensäädin 5

4 Välineet 5

4.1 Audi A6 vm. 2003 5
4.2 Raspberry Pi 4 Model B+ 6
4.3 Raspberry Pi 7” WVGA-kosketusnäyttö 6
4.4 Teraranger Evo 60m -etäisyysmittari 7
4.5 Benewake TF03-100 LiDAR 8
4.6 MKR CAN Shield 8
4.7 Raspberry Pi Relay Board 1.0 9
4.8 PCAN-View 9
4.9 PCAN-USB 10

5 Kehitysvaihe 11

5.1 Väyläviestit 11
5.1.1 Väyläviestien selvitys PCAN-View’n avulla 11
5.1.2 Väyläviestien lukeminen RasPilla 12
5.2 Tutkan lukeman vastaanottaminen RasPilla 13
5.2.1 Teraranger Evo 60m 13
5.2.2 Benewake TF03-100 14
5.3 Vakionopeudensäätimen ohjaus RasPilla 16

5.4 Käyttöliittymä 18
5.4.1 Konsolisovellus 18
5.4.2 Graafinen käyttöliittymä 18
5.5 Teline kosketusnäytölle 19
6 Testaus 21
6.1 Tutkien testaus 21
6.1.1 Teraranger Evo 60 m 21
6.1.2 Benewake TF03-100 LiDAR 22
6.2 Relekortin testaus 23
6.3 CAN-väyläkommunikaatio 25
6.4 Järjestelmän testaus käytännössä 25
7 Yhteenveto 26
Lähteet 28
Liitteet
Liite 1. Konsolisovelluksen lähdekoodi
Liite 2. Graafisen käyttöliittymän lähdekoodi

Lyhenteet

ACC	Adaptive Cruise Control, adaptiivinen vakionopeudensäädin
CAN	Controller Area Network, ajoneuvon tiedonsiirtoprotokolla
LiDAR	Light Detection and Ranging, optinen tutka, joka toimii näkyvän valon, lähi-infran tai ultravioletin alueella.
RasPi	Raspberry Pi, yhden piirilevyn tietokone
TF03	Benewake TF03-100 LiDAR

1 Johdanto

Tässä opinnäytetyössä rakennettiin adaptiivinen vakionopeudensäädin ajoneuvoon, jossa on varusteena perinteinen vakionopeudensäädin. Ajoneuvona käytettiin vuosimallin 2003 Audi A6:ta. Työ haluttiin toteuttaa välineillä, jotka ovat kenen tahansa saatavilla, joten kalliita ajoneuvoteollisuudessa käytettäviä välineitä ei käytetä.

Järjestelmän tarkoituksena on pitää automaattisesti edellä ajavaan ajoneuvoon kuljettajan valitsema vähimmäisetäisyys. Tämä toteutetaan siten, että auton etupuskuriin asennetaan etäisyysmittari, esimerkiksi optinen tutka (LiDAR), joka mittaa etäisyyttä edellä ajavaan ajoneuvoon. Lisäksi ajoneuvon väylältä poimitaan ajonopeustieto, jonka avulla etäisyys muutetaan sekunneiksi. Tämän tiedon avulla ohjataan ajoneuvon vakionopeudensäädintä minitietokone Raspberry Pin (RasPi) avulla niin, että kun etäisyys edellä olevaan ajoneuvoon lyhenee liian pieneksi, RasPi lähettää vakionopeudensäätimelle käskyn hidastaa. Kun välimatka on tarpeeksi pitkä, käskyn lähettäminen lopetetaan. Mikäli välimatka taas kasvaa suuremmaksi ja oma ajonopeus on alhaisempi kuin kuljettajan asettama, kiihdytetään, kunnes kyseinen nopeus on saavutettu tai etäisyys lyhenee kuljettajan valitsemaan minimiin.

Turvallisuussyistä järjestelmää ei kytketä lainkaan auton jarrujärjestelmään, vaan nopeudenmuutos toteutetaan ainoastaan vakionopeudensäätimen avulla, jolloin hidastuvuus rajoittuu moottorijarrutukseen ja näin vältytään tahattomilta äkkijarrutuksilta.

Kaikki kiinnitykset, liitännät ja kytkennät toteutetaan siten, että ne voidaan helposti poistaa ilman, että ajoneuvoon jää pysyviä jälkiä asennuksista.

2 Vakionopeudensäätimen historia

Tässä luvussa kuvataan lyhyesti vakionopeudensäätimen kehitysvaiheet sen keksimisestä adaptiivisen vakionopeudensäätimen kehitykseen.

2.1 Ensimmäiset vakionopeudensäätimet

Ajoneuvokäytössä vakionopeudensäätimiä on ollut 1900-luvun alusta asti, mutta nykyaikaisen vakionopeudensäätimen keksi Ralph Teetor vuonna 1948. Ensimmäinen versio toimi enemmänkin rajoittimena kuin vakionopeudensäätimenä, sillä se ainoastaan lisäsi kaasupolkimeen vastusta, kun ajoneuvo saavutti kuljettajan säätämän nopeuden estäen kiihdytyksen sitä suurempiin nopeuksiin. Tämä versio patentoitiin vuonna 1950. Myöhempiin versioihin Teetor lisäsi lukitustoiminnon, jonka avulla ajoneuvon nopeus pysyi aina samana, kunnes kuljettaja joko painoi jarrupoljinta tai kytki järjestelmän pois päältä. (The Sightless Visionary Who Invented Cruise Control 2018.)

2.2 Vakionopeudensäätimen yleistyminen

Vakionopeudensäätimen keksimisen jälkeen parin vuosikymmenen ajan vakionopeudensäädin oli kalliimpien automallien lisävaruste, jolla lisättiin ajomukavuutta. Vuoden 1973 öljykriisin, jonka aikana raakaöljyn hinta nelinkertaistui, vakionopeudensäädin yleistyi taloudellisena, polttoainetta säästävänä järjestelmänä. Nykyisin vakionopeudensäädin löytyy suurimmasta osasta ajoneuvoja. (Oil Embargo, 1973–1974.)

2.3 Adaptiivinen vakionopeudensäädin

Ensimmäinen askel adaptiivisen vakionopeudensäätimen yleistymiseen oli Mitsubishin vuonna 1991 esittelemä etäisyysvaroitin. Järjestelmää käytettiin Mitsubishin Debonair-automallissa ja se ainoastaan varoitti kuljettajaa edessä olevasta esteestä eikä osallistunut ajoneuvon nopeuden hallintaan lainkaan. Vuonna 1995 Mitsubishi toi markkinoille automallin Diamante, joka oli ensimmäinen sarjavalmisteinen ajoneuvo, jossa oli varusteena adaptiivinen vakionopeudensäädin. Järjestelmä käytti hyväkseen etupuskuriin

asennettua LiDARia sekä pientä kameraa taustapeilissä. Se pystyi havaitsemaan edessä olevan hitaamman ajoneuvon ja se hidasti auton nopeutta vastaamaan edellä ajavan nopeutta. Koska järjestelmä hyödynsi nopeuden hidastamiseen ainoastaan moottorijarrutusta, eikä tutkan kantama antanut järjestelmän toimia kuin enintään nopeuksissa 107 km/h, Mitsubishi asensi järjestelmän ainoastaan Japanin markkinoilla myytäviin autoihin. (The history of adaptive cruise control 2018.)

Vuonna 1999 Mercedes-Benz esitteli Distronic-järjestelmän, joka oli suunniteltu toimimaan suuremmissa nopeuksissa sekä huonommissa ajo-olosuhteissa kuin Japanissa kehitetyt. Aluksi järjestelmä oli saatavilla vain S-luokan autoihin eikä sitä ollut kytketty jarruihin lainkaan. Myöhemmin vuonna 2006 järjestelmään tuli päivitys, Distronic Plus, jonka myötä järjestelmä pystyi myös jarruttamaan automaattisesti. (Mercedes-Benz History of Innovation 2019.)

3 Vakionopeudensäätimen toiminta

Tässä luvussa tarkastellaan vakionopeudensäätimen ja adaptiivisen vakionopeudensäätimen toimintaa.

3.1 Perinteinen vakionopeudensäädin

Perinteinen vakionopeudensäädin kytketään päälle ajamalla haluttua nopeutta (yleensä yli 30 km/h) ja painamalla ohjaamossa sijaitsevaa kytkintä, jolloin vakionopeudensäädin pitää valitun nopeuden mahdollisimman tasaisesti. Vakionopeudensäädin saa ajoneuvon nopeustiedon esimerkiksi vetoakselin pyörimisnopeustiedosta tai nopeusmittarin anturilta. Mikäli ajoneuvon nopeus kasvaa säädetystä nopeudesta esimerkiksi alamäessä, vakionopeudensäädin pienentää nopeutta. Vastaavasti nopeuden pienentyessä esimerkiksi ylämäessä vakionopeudensäädin kasvattaa nopeutta vastaamaan säädettyä nopeutta. Nopeuden säätöä varten vakionopeudensäädin on kytketty kaasuläppään, jolla nopeutta säädetään myös normaalisti kaasupolkimen välityksellä.

Vanhemmissa autoissa kaasupoljin on kytketty kaasuläppään vaijerin välityksellä, ja näissä ajoneuvoissa myös vakionopeudensäädin säätää kaasuläpän asentoa vaijerin

välityksellä. Vaijeria liikutetaan solenoidin ja alipainekammion avulla. Tällaisissa järjestelmissä myös kaasupoljin liikkuu ylös ja alas vakionopeudensäätimen säätäessä ajonopeutta.

Uudemmissa ajoneuvoissa, joissa on ns. drive-by-wire-järjestelmä, kaasuläpän asentoa säädetään sähköisesti eikä kaasuläpän ja kaasupolkimen välillä ole mekaanista yhteyttä. Näissä ajoneuvoissa vakionopeudensäädin on yleensä moottorinohjausyksikön sisällä eikä kaasupoljin liiku vakionopeudensäätimen vaikutuksesta.

Perinteisissä vakionopeudensäätimen kytkimissä on yleensä seuraavanlaiset toiminnot: on (päälle), off (pois), set (nopeuden asetus), cancel (säädin väliaikaisesti pois päältä), resume (viimeisimmän nopeuden palautus), accelerate (nopeuden lisäys) ja coast (nopeuden vähennys). Jotkin näistä ovat usein yhdistetty saman kytkimen taakse, esimerkiksi set ja coast ovat usein samassa kytkimessä kuten myös resume ja accelerate. Myös on-, off- ja cancel-toiminnot löytyvät usein samasta kolmiasentoisesta liukukytkimestä. Tällöin kaikkien vakionopeudensäädinjärjestelmän toimintojen ohjaukseen riittää kolme kytkintä. Yleisimmät sijoituspaikat näille kytkimille ovat ohjauspyörä ja vilkkuviiksi.

Kytkeäkseen vakionopeudensäätimen toimintaan kuljettajan täytyy ajaa haluttua nopeutta, varmistaa, että on-off-cancel-kytkin on on-asennossa ja tämän jälkeen painaa set-kytkintä. Tämän jälkeen vakionopeudensäädin pitää valitun nopeuden mahdollisimman tarkasti, kunnes kuljettaja joko painaa cancel-kytkintä tai jarrupoljinta. Useimmissa käsivaihteistolla varustetuissa autoissa myös kytkinpolkimen painaminen kytkee vakionopeudensäätimen väliaikaisesti pois päältä. Kaasupolkimen painallus ei yleensä vaikuta vakionopeudensäätimen toimintaan, vaan kiihdytyksen jälkeen auton nopeus hidastuu säädettyyn nopeuteen.

Vakionopeudensäätimet yleensä muistavat viimeisimmän säädetyn nopeuden, mikäli järjestelmä on kytketty väliaikaisesti pois päältä. Ajoneuvolla voidaan kiihdyttää takaisin säädettyyn nopeuteen painamalla resume-kytkintä. Ajoneuvon virran sammuttaminen tai on-off-cancel-kytkimen siirtäminen off-asentoon kytkee koko järjestelmän pois päältä, jolloin viimeisin säädetty nopeus katoaa muistista.

3.2 Adaptiivinen vakionopeudensäädin

Adaptiiviset vakionopeudensäädinjärjestelmät (ACC) toimivat muuten samoin kuin perinteiset järjestelmät, mutta niissä on lisäksi auton etuosassa etäisyysanturi, jolla mitataan etäisyyttä edellä ajavaan ajoneuvoon. Mikäli edellä ajava ajoneuvo hidastaa, ACC havaitsee tämän muutoksen ja hidastaa samaan nopeuteen. Edellä ajavan auton kiihdyttäessä tai poistuessa tieltä, ACC kiihdyttää takaisin enintään kuljettajan säätämään nopeuteen. Ensimmäisissä ACC:issä ei ollut automaattista hätäjarrutustoimintoa, vaan hidastaminen oli ainoastaan moottorijarrutuksen varassa. Automaattivaihteisissa autoissa ACC pystyi antamaan käskyn vaihdelaatikolle vaihtaa pienemmälle vaihteelle moottorijarrutuksen tehostamiseksi, mutta sen nopeampaan hidastamiseen ne eivät pystyneet. Myöhemmissä versioissa on olemassa automaattinen hätäjarrutustoiminto, jolloin auto voi myös jarruttaa automaattisesti, mikäli pelkkä moottorijarrutus ei riitä hidastamaan tarpeeksi. Järjestelmissä on myös eroja autovalmistajien välillä.

4 Välineet

Tässä luvussa esitellään lyhyesti tässä projektissa käytetyt välineet. Niiden käytöstä käytännössä kerrotaan laajemmin myöhemmissä luvuissa.

4.1 Audi A6 vm. 2003

Tässä työssä oli käytössä Audi A6 vuosimallia 2003. Kyseinen ajoneuvo valikoitui projektissa käytettäväksi ajoneuvoksi, koska se sattui olemaan tätä opinnäytetyötä tekevän opiskelijan omistuksessa. Ajoneuvon tärkeimmät tekniset ominaisuudet ovat seuraavat:

- mallinimi: A6 2.0 CVT
- vuosimalli: 2003
- korimalli: C5
- vetotapa: etuveto
- vaihteisto: 01J Multitronic (CVT-automaatti)
- moottori: 2,0 l (1984 cm^3) 96 kW
- varusteena perinteinen vakionopeudensäädin

- ajoneuvoväylänä diagnostiikassa K-linja, mutta moottorinohjausyksikön, vaihteiston ja ESP:n välisessä tiedonsiirrossa käytössä CAN-väylä.

4.2 Raspberry Pi 4 Model B+

Raspberry Pi 4 Model B+ on viimeisin versio Raspberry Pista, joka on yhden piirilevyn tietokone. Siitä löytyy seuraavat ominaisuudet:

- suoritin: Broadcom BCM2711, neliytiminen Cortex-A72 (ARMv8) 64-bit SoC @ 1.5GHz
- keskusmuisti: 4GB LPDDR4-3200 SDRAM
- langattomat yhteydet: 2.4GHz ja 5GHz IEEE 802.11ac langaton verkkoyhteys, Bluetooth 5.0, BLE
- verkkoyhteys: Gigabit Ethernet (maksiminopeus 300 Mbps)
- 40-nastainen GPIO header
- 2 micro-HDMI-liitäntää
- 2 USB 2.0 -porttia ja 2 USB 3.0 -porttia
- CSI-kameraliitäntä Raspberry Pi -kameraa varten
- DSI-näyttöliitäntä Raspberry Pi -kosketusnäyttöä varten
- stereoääniliitäntä sekä komposiittivideoliitäntä
- micro SD -paikka
- 5V/3A DC -virransyöttö
- power-over-Ethernet (PoE) -tuki (tarvitsee erillisen PoE-hatun).

4.3 Raspberry Pi 7” WVGA-kosketusnäyttö

Ohjauspaneelina projektissa käytettiin RasPin 7 tuuman kosketusnäyttöä, jossa on mm. seuraavat ominaisuudet:

- WVGA (800x480)
- ruudun koko: 155 x 86 mm
- näytön mitat: 194 x 110 x 20 mm
- tuki jopa 10 pisteen kosketukselle
- liitäntä: DSI-kaapeli
- virransyöttö 5 V joko RasPin GPIO-nastoista tai MicroUSB-liitännällä

4.4 Teraranger Evo 60m -etäisyysmittari

Tutkana tässä projektissa käytettiin aluksi Teraranger Evo 60m -etäisyysmittaria, jossa on seuraavat ominaisuudet:

- infrapunavalon lentoaikaan perustuva etäisyyden mittaus
- toiminta-alue 0,5–60 m
- tarkkuus ±4 cm (≤ 14 m), ±1,5 % (> 14 m)
- pieni ja kevyt (vain 9 grammaa)
- USB, I2C ja UART -käyttöliittymät
- ROS- ja Pixhawk-valmius
- toiminta myös hämärässä ja pimeässä
- silmille turvallinen.

Teraranger Evo 60m:n etäisyysmittaus perustuu infrapunavalon lentoaikaan. Lentoaika on se aika, joka infrapunavalolla kestää kulkea lähettimestä jonkin esineen kautta takaisin vastaanottimeen. Lähetin ja vastaanotin sisältyvät molemmat etäisyysmittariin, ja esine oli tässä tapauksessa edellä ajava ajoneuvo. Lentoajan ja valonnopeuden perusteella voidaan laskea esineen etäisyys mittarista.

Infrapunavaloa hyödyntävä etäisyysmittari sopi tähän projektiin paremmin kuin laseretäisyysmittari, koska sillä voidaan mitata etäisyyttä suuremmalta alueelta. Laseretäisyysmittareissa etäisyyden mittaus perustuu samaan lentoaikaan, mutta lasersäteellä voidaan mitata etäisyys vain yhdestä pienestä pisteestä. Infrapunavalon hajaantuessa suuremmalle alueelle saadaan mittaustuloksia samalla kertaa suuremmalta alueelta. 1 metrin etäisyydellä mitattava alue on n. 3 cm * 3 cm ja 10 metrin etäisyydellä 30 cm * 30 cm. Mitattavan alueen koko muuttuu suoraan verrannollisesti sen etäisyyteen mittarista. Etäisyysmittarin maksimietäisyydellä, 60 m päässä, mitattava alue on n. 1,8 m * 1,8 m. Mittari palauttaa tuloksena etäisyyden esineeseen millimetreinä joko teksti- tai binäärimuodossa.

Teraranger Evo 60m -etäisyysmittari korvattiin myöhemmässä vaiheessa Benewake TF03-100 LiDARilla.

4.5 Benewake TF03-100 LiDAR

Myöhemmässä vaiheessa projektia tutkana käytettiin Benewake TF03-100 LiDARia, jossa on seuraavat ominaisuudet:

- toiminta-alue 0,1–100 m
- tarkkuus ±10 cm (<10 m), 1 % (≥ 10 m)
- mittaustuloksen esitystarkkuus 1 cm
- näytteenottotaajuus säädettävissä 1–1000 Hz (oletuksena 100 Hz)
- käyttölämpötila -25…+60 °C (säilytys -40…+85 °C)
- kotelon suojausluokitus IP67
- valonlähde LED (aallonpituus 905 nm)
- valokeilan koko 0,5 °
- virrankulutus ≤0,9 W
- käyttöliittymä UART/CAN/IO
- paino 77 g ± 3 g.

Benewake TF03-100 on teollisuustason LiDAR, johon on ohjelmoitu algoritmeja kompensoimaan ulkopuolisia häiriöitä, kuten esimerkiksi valon heijastuksia, vesi- tai lumisadetta tai sumua. Sen toimintaparametrit ovat täysin käyttäjän muutettavissa ja tästä syystä se valikoituikin projektiin Terarangerin korvaajaksi. Tärkein parametri oli kiinteä näytteenottotaajuus, jonka ansiosta TF03-100 saatiin synkronoitua samalle taajuudelle ajoneuvoväylällä kulkevan ajonopeustiedon kanssa.

4.6 MKR CAN Shield

Ajoneuvon väyläkommunikointiin käytettiin alun perin Arduinolle tarkoitettua MKR CAN Shieldiä, jota pystyy käyttämään myös RasPilla SPI-väylän avulla. Siitä löytyy seuraavat ominaisuudet:

- protokolla: CAN-väylä
- käyttöliittymä: SPI-väylä
- piirin käyttöjännite: 3,3 V
- väyläohjain: MCP2515

- lähetin/vastaanotin: NXP TJA1049
- Buck-hakkuri: TPS54232
- V_{in} (ruuviliitin): 7–24 V
- V_{in} (pinneistä): 5 V
- yhteensopivuus: MKR-koko
- kytkimellä varustettu 120 Ω:n päätevastus.

4.7 Raspberry Pi Relay Board 1.0

Ajoneuvossa olemassa olevan vakionopeudensäätimen ohjaukseen käytettiin SeeedStudion valmistamaa Raspberry Pi Relay Board 1.0 -relekorttia, jossa on seuraavat ominaisuudet:

- I2C-käyttöliittymä
- ruuviterminaalit releille
- LED-merkkivalot jokaiselle releelle
- COM-, NO- ja NC-nastat jokaiselle releelle
- 4 korkealaatuista relettä (10 A 30 VDC).

Relekortti toimii RasPin ns. hattuna eli se asetetaan suoraan RasPin päälle. Datayhteys relekortin ja RasPin välillä toimii GPIO-pinnien kautta, ja niitä varten relekortin pohjassa on naaraspuoliset nastat, jotka kytkeytyvät suoraan RasPin pinneihin. Relekortin päällä on myös läpivientinä urospuoliset nastat, joten RasPin GPIO-pinnejä pystyy käyttämään relekortin läpi sen ollessa paikallaan RasPin päällä. Relekortin I2C-osoitetta pystyy myös tarvittaessa vaihtamaan relekortin pinnassa olevien kytkimien avulla. Oletuksena osoite on 0x20.

4.8 PCAN-View

Projektin alkuvaiheessa tutkittiin ajoneuvon CAN-väylällä kulkevia viestejä PEAK-Systemin valmistamalla PCAN-View-ohjelmistolla, josta löytyy seuraavat ominaisuudet:

- tuki CAN-spesifikaatioille 2.0 A/B sekä FD
- bittinopeus valittavissa 1 Mb/s asti (CAN FD:llä jopa 12 Mb/s)

- mukautetut bittinopeudet valittavissa
- vain kuuntelu -tila
- manuaalinen sekä ajoittainen viestien lähetys vähintään 1 ms resoluutiolla
- viestien vastaanotto vähintään 100 µs:n resoluutiolla
- vastaanotettujen viestien tallennus
- CAN ID:n näyttö joko heksadesimaaleina tai desimaaleina
- datatavujen näyttö joko heksadesimaaleina, desimaaleina tai ASCII-muodossa
- virheviestien vastaanotto
- CAN-väyläohjaimen resetointi.

4.9 PCAN-USB

PCAN-View-ohjelmaa varten tarvittiin myös PEAK-Systemin valmistama PCAN-USB, jolla tietokone saatiin yhdistettyä ajoneuvon CAN-väylään. Siitä löytyy seuraavat ominaisuudet:

- USB-liitäntä (yhteensopiva USB 1.1:n, USB 2.0:n ja USB 3.0:n kanssa)
- high-speed CAN-yhteys (ISO 11898-2 -protokolla)
- bittinopeus valittavissa väliltä 5 kb/s–1 Mb/s
- aikaleiman resoluutio n. 42 µs
- yhteensopiva CAN spesifikaatioiden 2.0A (11-bittinen ID) ja 2.0B (29-bittinen ID) kanssa
- yhteys CAN-väylälle 9-pinnisellä D-SUB-liittimellä (CiA® 303-1:n mukaisesti)
- CAN-väyläohjain: NXP SJA1000, 16 MHz:n kellotaajuus
- CAN-väylälähetin/-vastaanotin: NXP PCA82C251
- valinnainen 5 V:n jännitesyöttö mahdollista ulkoista väylämuunninta varten
- galvaanisesti eristetty CAN-väylästä 500 V:iin asti
- virransyöttö USB-liitännän kautta
- toimintalämpötila -40–+85 °C

5 Kehitysvaihe

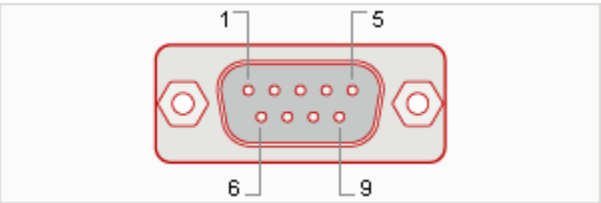
Tässä luvussa käsitellään koko kehitysvaihe alusta siihen asti, kunnes tuote on valmis testattavaksi.

5.1 Väyläviestit

5.1.1 Väyläviestien selvitys PCAN-View'n avulla

Väyläviestien selvityksessä lähdettiin liikkeelle selvittämällä mistä ajoneuvossa löytyy CAN-väylä. Tässä käytettiin apuna Metropoliassa käytössä olevaa Boschin ESI[tronic] 2.0 -ohjelmaa, ja sieltä löytyviä kytkentäkaavioita. Niiden perusteella selvisi, että helpoin paikka, josta väylälle pääsee kytkeytymään, olisi moottorinohjausyksikön johtosarja. Moottorinohjausyksikkö sijaitsee tässä automallissa moottoritilan vasemmassa reunassa tuulilasinpyyhkijöiden alapuolella olevassa muovikotelossa. Tästä kotelosta on kätevästi suora yhteys kuljettajan jalkatilassa sijaitsevaan relerasiaan, joten johtimet saatiin sieltä kuljetettua helposti matkustamon puolelle. Kytkentäpiste maajohdolle löytyi niin ikään kuljettajan jalkatilasta, relerasian vieressä olevasta maapisteestä. Tässä ajoneuvossa CAN-väylän signaalimaa on yhdistetty ajoneuvon runkomaahan, joten maadoitus voitiin hoitaa tätä kautta.

Haaroitusta varten kuorittiin moottorinohjausyksikön ja ESP-ohjainlaitteen välisistä väyläjohtimista n. 1 cm:n pituinen alue, johon toiset väyläjohtimet juotettiin kiinni. Väyläjohtimina käytettiin 0,75 mm^2:n paksuisia monisäikeisiä kuparijohtimia, jotka kierrettiin parikaapeliksi. Johtimien toiseen päähän juotettiin naaraspuolinen 9-nastainen D-SUB-liitin PCAN-USB:tä varten. Liittimen pinnijärjestys näkyy kuvassa 1.



2 – CAN-L
3 – Maa
7 – CAN-H

Kuva 1. PCAN-USB:n D-SUB-liittimen pinnijärjestys

Kun tarvittavat liitännät oli saatu valmiiksi, autoa nostettiin nosturilla sen verran, että saatiin renkaat ilmaan. Näin voitiin turvallisesti etsiä nopeustietoa väyläviesteistä auton ollessa paikallaan. Varmuuden vuoksi myös ESP-järjestelmä kytkettiin pois päältä, jotta nopeutta voitiin nostaa sen puuttumatta peliin. Tämän jälkeen auto käynnistettiin, laitettiin vaihde päälle ja sopivin väliajoin nostettiin jalka jarrulta, annettiin pyörien pyöriä hetken, minkä jälkeen painettiin jarrua. Samalla tarkkailtiin PCAN-View'n ruudulta, mikä tai mitkä viestit muuttuvat sen näköisesti, että ne voisivat sisältää ajonopeustiedon. Melko nopeasti havaittiin, että useammassakin viestissä jokin tavuista muuttui näyttäen siltä, että se sisältää ajonopeustiedon, mutta kaikissa luku näytti olevan selvästi oikeaa nopeutta alempi. Ajonopeustieto löydettiin seuraavista viesteistä:

- CAN ID 288, 4. tavu
- CAN ID 1A0, 4. tavu
- CAN ID 4A0, 4. tavu
- CAN ID 5A0, 3. tavu (tässä lukema pienempi kuin muissa)
- CAN ID 320, 7. tavu (tässä lukema suurempi kuin muissa).

Vakionopeudensäätimen painikkeiden tieto löytyi väyläviesteistä CAN ID:llä 38A. Viestin toisessa tavussa olevasta tiedosta pystyi päättelemään, mikä nappi on painettuna, mutta ensimmäinen ja kolmas tavu olivat ns. syklaavia tavuja. Tämä tarkoittaa, että tavuissa oleva tieto kiertää tiettyä sykliä ja niiden sisältö on joka viestissä eri. Tästä syystä vakionopeudensäädintä ei voitu ohjata väylän kautta vaan piti miettiä muita ratkaisuja.

5.1.2 Väyläviestien lukeminen RasPilla

RasPilla väyläviestit luettiin alun perin Arduinolle valmistetun MKR CAN Shieldin avulla. Kyseistä CAN Shieldiä pystyttiin hyvin käyttämään myös RasPin kanssa, koska se toimii SPI-väylän välityksellä.

Kehitysvaiheessa havaittiin, että CAN-väyläkommunikaatiota varten RasPille on jo luotu valmis kirjasto can-utils, josta löytyy valmiit Python-komennot viestien lukemista varten. Kirjaston ansiosta CAN-väyläviestien lukeminen Python-ohjelmalla oli melko helppoa. Kun oli saatu selville CAN-ID:t, jotka sisältävät ajonopeustiedon, voitiin viesteistä suodattaa pois kaikki tarpeettomat viestit. Valittiin käytettäväksi CAN ID 320, koska siinä

lukema oli lähinnä oikeaa nopeutta, vaikka siinäkin se oli pienempi kuin todellinen nopeus. Tätä nopeustietoa verrattiin älypuhelimen GPS-vastaanottimen nopeustietoon ja todettiin, että väylällä oleva nopeuslukema täytyy kertoa 1,176:lla, jotta saadaan oikea ajonopeus.

5.2 Tutkan lukeman vastaanottaminen RasPilla

5.2.1 Teraranger Evo 60m

Projektin alkuvaiheessa käytössä ollut tutka, Teraranger Evo 60m, hyödyntää lukeman lähetykseen sarjakommunikaatiota. Tähän tarkoitukseen oli valittavana joko USB- tai I2C/UART-yhteys. Valinta tapahtui valitsemalla ostovaiheessa tutkalle taustalevy, jossa on tarvittavan yhteyden mahdollistava liitäntä. Molemmat yhteystavat perustuvat sarjakommunikaatioon, joten päädyttiin valitsemaan USB-yhteys, koska se on monikäyttöisempi. USB-yhteyden avulla tutkaa voitiin testata kannettavan tietokoneen tai älypuhelimen kanssa.

RasPilla tutkan lukema vastaanotettiin Python-ohjelmointikielellä kirjoitetulla ohjelmalla, joka avaa sarjaportin ja valmistelee tutkan lähettämään tietoa tekstimuodossa, jolloin se on helpointa tulkita. Sarjaportin avaamiseen käytettiin seuraavia määrittelyjä: bittinopeus 115200 b/s, 8 databittiä, 1 stoppibitti ja ei pariteettia. Lisäksi tutkalle piti sarjaportin avauksen jälkeen lähettää heksadesimaalimuodossa viesti 00 11 01 45, jotta se alkaa lähettää lukemia tekstimuodossa. Binäärimuotoa varten pitää lähettää viesti 00 11 02 4C. Tekstimuoto valittiin, koska tällöin tutka lähettää kerrallaan yhden lukeman, joka päättyy rivinvaihtomerkkiin. Tällöin viesteistä on helppo poimia yksi lukema ja muuntaa se desimaalinumeromuotoon.

Mikäli kohde on liian lähellä (alle 0,5 m) tutka lähettää viestin ”-Inf”. Mikäli taas kohde on liian kaukana (yli 60 m), tutka lähettää viestin ”+Inf”. Jos taas mittausta ei jostain syystä pystytä suorittamaan ollenkaan, tutka lähettää viestin ”-1”.

5.2.2 Benewake TF03-100

Myöhemmässä vaiheessa projektia käyttöön otettu Benewake TF03-100 LiDAR kytkeytyy RasPiin UART-yhteyden välityksellä. TF03:ssa oli valmiina datakaapeli, jonka päässä oli 7-nastainen Molex PicoBlade -liitin. Tällaisella liittimellä varustettua datakaapelia ei löytynyt valmiina, joten sellainen piti kasata itse. Kaapeliksi tutkalta ohjaamoon valittiin suojattu datakaapeli Tasker C508 8 x 0,12 mm^2. Datakaapeli oli värikoodattu samalla tavalla kuin TF03:sta lähtevä kaapeli, joten johtimet oli helppo kytkeä oikein molemmissa päissä. Ylijäänyt kahdeksas johdin jätettiin liittämättä. Johtimien kytkentäjärjestys näkyy taulukossa 1.

Taulukko 1. TF03 johtimien kytkentäjärjestys

Numero	**Väri**	**Johdin**	**Toiminto**
1	Punainen	VCC	Käyttöjännite
2	Valkoinen	CAN_L	CAN_L-johdin
3	Vihreä	CAN_H	CAN_H-johdin
4	Sininen	GPIO	IO-ulostulo
5	Ruskea	TTL_RXD	UART-vastaanotto
6	Keltainen	TTL_TXD	UART-lähetys
7	Musta	GND	Maadoitus

Kommunikointiin RasPin kanssa tarvittiin vain käyttöjännite, UART-vastaanotto ja -lähetys sekä maadoitus. Loput johtimista jätettiin kytkemättä. Luonnollisesti TF03:n UART-vastaanottojohdin kytkettiin RasPin UART-lähetysnastaan ja päinvastoin.

Koska UART-kommunikaatioyhteys toimii hyvin samalla tavalla kuin USB-yhteys, voitiin viestien vastaanottoon käyttää samaa koodia kuin Teraranger Evo 60m:n kanssa. Ainoastaan vastaanotetun viestin käsittelyä piti muuttaa hieman. Sarjaportin avaamiseen käytettiin seuraavia määrittelyjä: bittinopeus 115200 bit/s, 8 databittiä, 1 stoppibitti ja ei pariteettia. Yhteyden muodostuksen jälkeen TF03 alkaa lähettää mittaustuloksia oletuksena 100 Hz:n taajuudella. Viestit ovat 9 tavun mittaisia, ja niissä on taulukon mukainen rakenne.

Taulukko 2. TF03:n lähettämän viestin rakenne

Datatavu	**Määritelmä**	**Sisältö**
Tavu 0	Otsikko	0x59
Tavu 1	Otsikko	0x59
Tavu 2	DIST_L	etäisyyden ensimmäiset 8 bittiä
Tavu 3	DIST_H	etäisyyden jälkimmäiset 8 bittiä
Tavu 4	Varattu tavu	/
Tavu 5	Varattu tavu	/
Tavu 6	Varattu tavu	/
Tavu 7	Varattu tavu	/
Tavu 8	Tarkistussumma	Checksum8 % 256 Tarkistussumma = (tavu 0 + tavu 1 + … + tavu 7) % 256

TF03 lähettää mittaustuloksen senttimetreinä heksadesimaalimuodossa. Koska suurin mahdollinen arvo 8-bittiselle heksadesimaaliluvulle on FF, joka on desimaaleina 255, mahtuu tavuun 2 vain maksimissaan 255 cm:n mittaustulos. Näin ollen tavussa 3 on aina luvun 256 monikerta, johon lisätään tavussa 2 oleva luku. Esimerkiksi etäisyys 10 m näkyy viestissä muodossa E8 03 (232 cm + 3 * 256 cm = 1000 cm = 10 m).

Tarkistussumma saadaan laskemalla yhteen tavujen 0–7 arvot ja ottamalla siitä 256:lla jakamisen jakojäännös. Esimerkiksi edellä mainitun 10 m mittaustuloksen sisältävän viestin tarkistussumma olisi 9D (89 + 89 + 232 + 3 = 413, jonka 256:lla jakamisen jakojäännös on 157, joka muunnettuna heksadesimaaliluvuksi on 9D). Viesti näyttäisi siis kokonaisuudessaan tältä: 59 59 E8 03 00 00 00 00 9D.

RasPilla tutkan lukemaa vastaanottaessa Python-ohjelmalla kirjoitettiin koodiin osio, joka lukee sarjaportista 9 tavua, laskee edellä mainitulla tavalla viestille tarkistussumman ja vertaa sitä viestin viimeisessä tavussa olevaan tarkistussummaan. Mikäli tarkistussummat täsmäävät, otetaan viestistä tavut 2 ja 3 ja muunnetaan niiden sisältö yhdeksi desimaaliluvuksi. Jos taas tarkistussummat eivät täsmää, jätetään viesti käsittelemättä. Tarkistussumman avulla voidaan ohittaa esimerkiksi synkronointihäiriöstä johtuvat virheelliset viestit.

Tutkan lähettämän valokeila on hyvin kapea, joten suuntaamisessa piti olla hyvin tarkkana. 100 metrin päässäkin sen leveys on 100 cm ja korkeus vain 28 cm. Sivusuunnassa se sallii siis pienen poikkeaman keskilinjasta, mutta korkeussuunnassa hyvin vähän.

Käytännössä tämä tarkoittaa, että esimerkiksi 1,5 m korkean auton ollessa 100 m päässä vain 0,5°:n muutos tutkan asennossa aiheuttaa sen, että sen valokeila ei osu kyseiseen autoon. Jotta tutka toimisi luotettavasti edes 50 m:n etäisyydellä, täytyy sen olla suunnattu 1°:n tarkkuudella vaakasuoraan. Koska tutka ei lähetä minkäänlaista näkyvää valoa, piti suuntaus suorittaa siten, että pysäköitiin auto mahdollisimman tasaiselle alustalle (esimerkiksi suuri parkkipaikka), asetettiin jokin mitattava este (esimerkiksi toinen auto) mahdollisimman kauas suoraan auton eteen, mitattiin etäisyys esteeseen toisella etäisyysmittarilla ja suunnattiin tutkaa, kunnes sen lukema näytti samaa kuin toisella etäisyysmittarilla.

5.3 Vakionopeudensäätimen ohjaus RasPilla

Koska väyläviestejä tutkiessa havaittiin, että CAN-väylällä kulkevassa vakionopeudensäätimen viestissä kaksi tavuista olivat ns. syklaavia tavuja, piti keksiä jokin muu keino vakionopeudensäätimen ohjaamiseen. Tutkittiin kytkentäkaavioita ja löydettiin vilkkuviiksen juuresta vakionopeudensäätimen liitin. Mitattiin johtimien jännitteitä eri tilanteissa ja eri painikkeita painettaessa ja havaittiin taulukon mukaiset jännitteet. Taulukossa käytetty lyhenne CC tarkoittaa vakionopeudensäädintä.

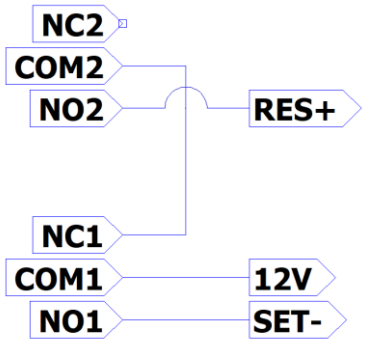
Taulukko 3. Vakionopeudensäätimen liittimen johtimien jännitteet

Johdin	CC off	CC on	CC cancel	CC set	CC resume
pun-har	0	12	0	12	12
valk	12	12	12	12	12
pun-kelt	0	0	0	12	0
must-valk	0	12	12	12	12
sin-lila	0	0	0	0	12

Taulukosta voidaan havaita, että punakeltaiseen johtimeen tulee jännite vain, kun painetaan set-painiketta, ja sinililaan, kun painetaan resume-painiketta. Tästä pääteltiin, että napit toimivat kytkiminä, jotka kytkevät 12 V tarvittavaan johtimeen silloin, kun nappia painetaan. Tämä testattiin haaroittamalla punaharmaasta, punakeltaisesta sekä sinililasta johtimesta uudet johtimet ja kytkettiin punaharmaassa johtimessa oleva 12 V vuoroin punakeltaiseen ja sinililaan johtimeen ja tutkimalla väyläviestejä. Havaittiin, että väy-

läviesti, joka sisältää tiedon vakionopeudensäätimen nappien painalluksista, muuttuu samalla tavalla kuin oikeastikin nappeja painettaessa. Jotta vakionopeudensäädintä voidaan ohjata tällä kyseisellä tavalla RasPilla, tarvitaan releitä, koska RasPi pystyy käsittelemään vain korkeintaan 5 V:n jännitteitä.

Ratkaisuna päädyttiin käyttämään SeeedStudion valmistamaa Raspberry Pi Relay Board 1.0 -relekorttia, jonka neljää relettä voidaan ohjata Python-koodilla. Relekortin kaikista releistä löytyy normally open- ja normally closed -tila. Tämä tarkoittaa sitä, että jokaisessa releessä on kolme terminaalia merkinnöiltään COM, NC ja NO. Normally closed -tilassa, kun rele on pois päältä, COM- ja NC-terminaalit ovat yhdistetty. Kun rele kytketään päälle, COM- ja NO-terminaalit kytketään yhteen. Tällöin NC-terminaali ei ole yhdistettynä. Vakionopeudensäädin kytkettiin relekorttiin siten, että punaharmaa johdin, johon tulee 12 V, kytkettiin releen 1 COM-terminaaliin ja edelleen sen NC-terminaalista releen 2 COM-terminaaliin. Tällöin johdinta ei tarvitse haaroittaa ja releen 1 ollessa pois päältä 12 V kulkee sen NC-terminaalin kautta releelle 2. Näin relettä 2 voidaan käyttää vain silloin, kun rele 1 on pois päältä. Tällä myös varmistetaan se, ettei vahingossa paineta molempia nappeja samaan aikaan. Releen 1 NO-terminaaliin kytkettiin punakeltainen johdin, jolloin se toimii vakionopeudensäätimen set-nappina, jolla vähennetään nopeutta. Vastaavasti releen 2 NO-terminaaliin kytkettiin sinilila johdin, jolloin se toimii resume-nappina, jolla lisätään nopeutta. Relekortin kytkentäkaavio näkyy kuvassa 2.



Kuva 2. Relekortin kytkentäkaavio

Tällä kytkennällä varmistettiin myös se, että jos jostain syystä molemmat releet ovat auki samaan aikaan, kytkeytyy virta ainoastaan set-kytkimeen, jolloin auto hidastaa. Tällä estetään auton tahaton kiihdyttäminen vikatilanteessa.

5.4 Käyttöliittymä

Tavoitteena projektissa oli kehittää graafinen käyttöliittymä, jota voidaan käyttää kosketusnäytön välityksellä. Tässä luvussa käsitellään sen kehitystä. Ohjelmointikieleksi valittiin Python-kieli sen yksinkertaisuuden vuoksi.

5.4.1 Konsolisovellus

Konsolisovelluksen kehittäminen oli melko yksinkertaista ja ohjelma oli helppo saada toimimaan, mutta toimintojen ohjaus on sillä hyvin hankalaa. Konsolisovelluksen ulkoasu koostuu ainoastaan tekstipohjaisista tulosteista eikä siinä ole lainkaan graafisia elementtejä. Lisäksi syötteiden vastaanottamista varten ohjelman suoritus täytyisi keskeyttää, joten esimerkiksi etäisyyden tai ajonopeuden valinta konsolisovelluksessa ei ole kovin käytännöllistä. Tästä syystä konsolisovellusta käytettiin kehitysvaiheessa lähinnä tutkan toiminnan testaamiseen. Konsolisovelluksen lähdekoodi löytyy liitteestä 1.

5.4.2 Graafinen käyttöliittymä

Graafista käyttöliittymää suunniteltaessa pyrittiin pitämään sovelluksen ulkoasu yksinkertaisena ja helppolukuisena. Myös painikkeet suunniteltiin helppokäyttöisiksi, jotta niiden käyttö ei vaarantaisi liikenneturvallisuutta. Python-ohjelmointikieltä käytettäessä graafinen käyttöliittymä luotiin hyödyntäen Tkinter-kirjastoa. Tkinterillä käyttöliittymän elementit luodaan tekstipohjaisesti, jolloin lopputulos täytyy tarkistaa käynnistämällä sovellus. Ohjelman suorituksen sujuvuuden vuoksi käytettiin Pythonin Multiprocessing-kirjastoa, jonka avulla jokainen toiminto, kuten tutkan lukeman vastaanotto, nopeuden lukeminen CAN-väylältä, relekortin ohjaus ja kaikki laskutoimitukset pyörivät taustalla kukin omina prosesseinaan ja niitä ohjattiin pääohjelmasta.

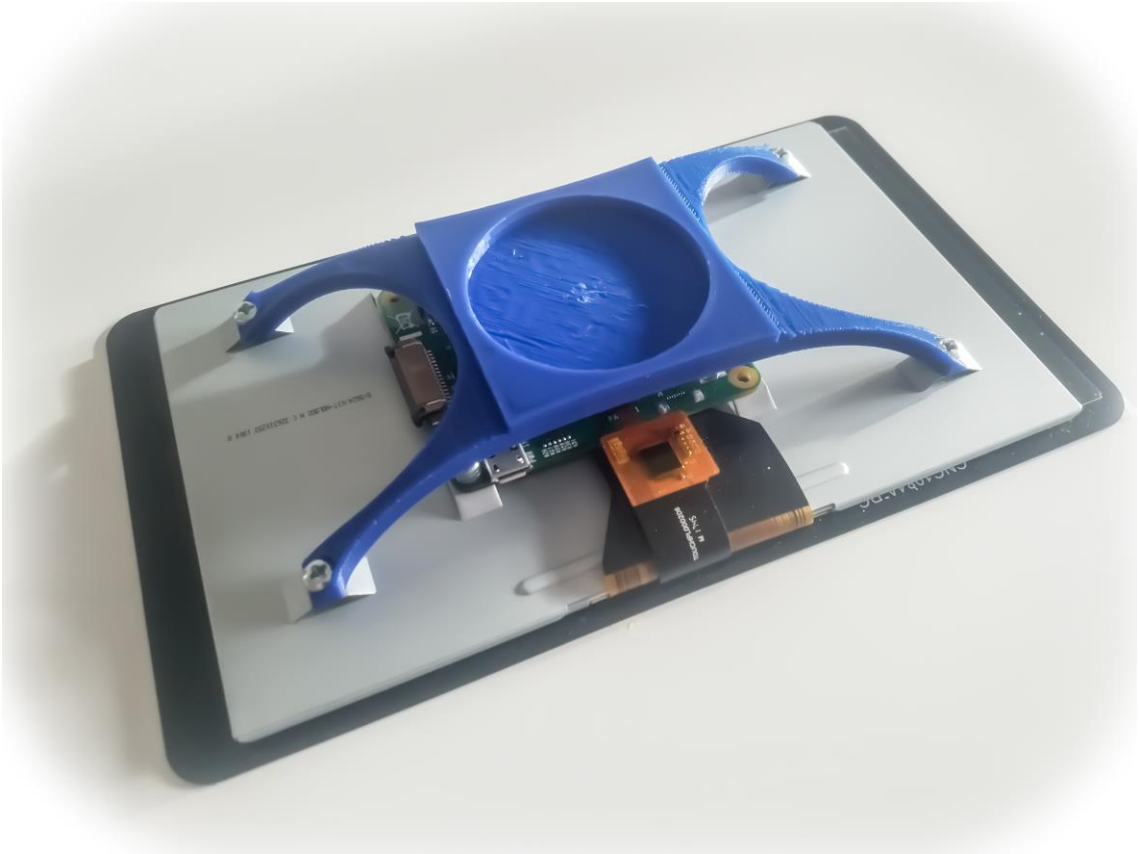
Sovelluksen näytölle valittiin näytettäväksi oma ajonopeus sekä etäisyys edellä ajavaan ajoneuvoon metreinä ja sekunteina. Lisäksi luotiin painikkeet etäisyyden ja nopeuden valintaa varten sekä sovelluksesta poistumisnäppäin. Etäisyydenvalitsimen vaihtoehdoiksi valittiin 1,0 s, 1,5 s ja 2,0 s ja nopeudenvalitsimen vaihtoehdoiksi 40–100 km/h 10 km/h välein sekä 120 km/h. Esimerkkikuva sovelluksen näytöstä näkyy kuvassa 3. Sovelluksen lähdekoodi löytyy liitteestä 2.

Kuva 3. Esimerkkikuva sovelluksen näytöstä.

5.5 Teline kosketusnäytölle

Projektissa käytetty kosketusnäyttö piti saada asennettua autossa johonkin, mistä se on helposti näkyvillä ajaessa ja sen painikkeet ovat myös helposti käytettävissä liikenneturvallisuutta vaarantamatta. Tähän tarkoitukseen löytyi sopivasti Garmin-merkkisestä navigaattorista ylijäänyt imukuppiteline, johon 3D-tulostettiin sopiva pidike, joka ruuvattiin kiinni kosketusnäyttöön. Näin näytön voi myös ottaa helposti mukaansa esimerkiksi yöksi. Kuvassa 4 näkyy teline 3D-mallin renderöintinä ja kuvassa 5 valmiina 3D-tulosteena.

Kuva 4. RasPin kosketusnäytön teline 3D-mallin renderöintinä.



Kuva 5. RasPin kosketusnäytön teline kiinnitettynä kosketusnäyttöön.

3D-tulostukseen käytettiin Metropolian Leiritien toimipisteestä löytyvää Ultimaker 3 Extended -mallista 3D-tulostinta. Materiaalina tulosteessa käytettiin PLA:ta.

6 Testaus

Tässä luvussa kuvataan tuotteen testausta kehitysvaiheessa komponenteittain sekä valmiina tuotteena.

6.1 Tutkien testaus

6.1.1 Teraranger Evo 60 m

Projektiin valittiin tutkaksi ensin Teraranger Evo 60m, koska sen lähettämä valokeila oli laajempi verrattuna TF03:een ja siinä oli riittävä kantama. Tutka oli melko helppo suunnata riittävällä tarkkuudella eteenpäin, mutta toimintaa testatessa havaittiin siinä huomattavia puutteita.

Ensimmäinen havaittu huono puoli Terarangerissä oli, että se toimii ns. free ranging -periaatteella, jonka vuoksi sen viestien lähetystaajuus vaihteli välillä 1 – 240 Hz riippuen siitä, kuinka kaukana edessä oleva esine on tai onko sitä ollenkaan. Free ranging -periaate tarkoittaa sitä, että se lähettää seuraavan mittaussignaalin vasta sitten, kun se on vastaanottanut signaalin edellisestä mittauksesta. Tästä syytä mittaustaajuus voi vaihdella suurestikin. Tutkaan on ohjelmoitu aikakatkaisu, jonka pituus vaikutti olevan yksi sekunti. Mikäli lähetettyä signaalia ei vastaanoteta tässä ajassa, lähetetään viesti ”-1” ja aloitetaan uusi mittaus. Näin ollen tutkan mittaustaajuus vaihtelee jatkuvasti. Tästä koitui ongelmia kehitysvaiheessa, koska tutkan lukemaa ja ajonopeustietoa ei voitu vastaanottaa synkronoidusti. Väyläviestejä tutkiessa havaittiin, että väyläviesti, joka sisältää ajonopeustiedon, lähetetään väylälle 20 ms:n välein eli 50 Hz:n taajuudella. Tämän takia silloin, kun tutkan mittausalueella ei ollut kohteita ja se lähetti lukemia 1 Hz:n taajuudella, nopeustietoja jäi puskuriin 49 viestiä sekunnissa. Kun jokin kohde tuli tutkan mittausalueelle, ohjelma jatkoi väyläviestien vastaanottamista puskurista aloittaen vanhimmasta viestistä, jolloin nopeustieto oli vanhentunutta. Vastaavasti, jos kohde oli tutkasta alle metrin päässä ja se lähetti lukemaa 240 Hz taajuudella, etäisyystietoja jäi puskuriin 190

viestiä sekunnissa. Tällöin taas tutka ”jäi jälkeen” ja etäisyystieto oli vanhentunutta. Ratkaisuksi tähän kokeiltiin tyhjentää sarjaportin puskuri aina, kun yksi viesti on käsitelty. Tämä auttoi poistamaan viesteistä vanhentuneet tiedot, jolloin tutkalta vastaanotettu etäisyystieto oli aina tuorein mahdollinen. Tämä ei kuitenkaan auttanut tilanteessa, jossa tutkan kantaman alueella ei ollut esinettä ja sen lähetystaajuus hidastui 1 Hz:iin. Tällöin koko ohjelman suoritus hidastui ja väyläviestejä kasaantui puskuriin. Esimerkiksi, kun testattiin toimintoa, jossa nopeuden hidastuksen jälkeen kiihdytetään takaisin valittuun maksimiajonopeuteen, ehti ohjelma antaa nopeuden kiihtyä noin 110 km:iin /h, vaikka maksiminopeudeksi oli valittu 80 km/h. Tämä johtui siitä, että tutkan lähetystaajuuden ollessa niin alhainen väyläviestejä kasaantui puskuriin niin paljon, että vasta nopeuden ollessa 110 km/h ohjelma vastaanotti väyläviestin, jossa ajonopeus oli vähintään 80 km/h.

Toinen huono puoli tässä tutkassa oli, että se vastaanotti virhetuloksia esimerkiksi kaistan yläpuolisista opasteista. Varsinainen tutkan lähettämän valokeilan koko on vain noin 1,7°, joten sen ei pitäisi osua kaistan yläpuolisiin opasteisiin. Häiriölukemat johtuivat siis todennäköisesti pienistä hajaheijastuksista, jotka heijastuvat opasteiden heijastinpinnasta takaisin tutkan sensoreihin tarpeeksi voimakkaasti ja tutka tulkitsi nämä varsinaisiksi mittaustuloksiksi.

Kolmas ongelma tutkan toiminnassa oli, että se toimi hyvin epäluotettavasti huonoissa sääolosuhteissa. Vesisateessa tai sen jälkeen tien pinnan ollessa märkä auton renkaista nousee ilmaan paljon pieniä vesipisaroita, jotka estivät tutkaa mittaamasta suurempia etäisyyksiä kuin noin 6 m. Tästä syystä projektiin lähdettiin etsimään toimivampaa tutkaa.

6.1.2 Benewake TF03-100 LiDAR

TF03 valittiin tähän työhön Terarangerin tilalle ensisijaisesti sen kiinteän lähetystaajuuden takia. Siinä viestien lähetystaajuus voidaan valita itse väliltä 1–1000 Hz. Heti alkuun oletustaajuudella 100 Hz testattuna havaittiin, että ohjelma toimi paljon sujuvammin TF03:n kanssa. Lähetystaajuutta yritettiin muuttaa 50 Hz:iin vastaamaan väyläviestien lähetystaajuutta, mutta se osoittautui yllättävän hankalaksi eikä 100 Hz:n lähetystaajuudesta vaikuttanut koituvan minkäänlaisia ongelmia, joten se jätettiin muuttamatta.

TF03 toimi huomattavasti paremmin myös huonoissa sääolosuhteissa sen sisäänrakennettujen algoritmien ansiosta. Ainoa mittatarkkuuteen merkittävästi vaikuttava asia oli sen valokeilan kapeus. Esimerkiksi 60 m:n etäisyydellä TF03:n valokeilan koko on vain 0,6 m * 0,17 m, kun Terarangerissä se oli 1,8 m * 1,8 m. Tästä huolimatta tutka saatiin toimimaan silmämääräisellä suuntauksella n. 40 m:iin asti, mikä mahdollisti adaptiivisen vakionopeudensäätimen testauksen lyhyimmillä etäisyysasetuksilla.

Alkuvaiheessa tutkan testausta tutka toimi täysin moitteettomasti ja näytti etäisyyksiä ihan oikein n. 60 m:iin asti. Myöhemmin kuitenkin havaittiin, että tutkan kantama pikkuhiljaa lyheni niin, että se lopulta pystyi enää mittaamaan etäisyyksiä n. 7 m:iin asti. Tutka oli ollut useamman viikon asennettuna kiinteästi auton etupuskuriin ja tänä aikana ulkolämpötila oli vaihdellut välillä -8…+7 °C, useimmiten sen ollessa n. +2 °C. Kokeiltiin irrottaa tutka autosta ja vietiin sisätiloihin, jossa se sai lämmetä yön yli. Seuraavana päivänä testatessa tutka näytti taas lukemia oikein ainakin n. 50 m:iin asti. Tästä pääteltiin, että lämpötilan lasku aiheuttaa tutkan kantaman pienenemisen, vaikka valmistaja kertookin sen käyttölämpötilan olevan -25…+60 °C. Ratkaisuksi mietittiin lämmittimen rakentamista tutkalle, mutta sen toteutus päätettiin jättää myöhemmäksi.

6.2 Relekortin testaus

Relekorttia testatessa ei havaittu minkäänlaisia ongelmia sen toiminnassa. Sen ohjaukseen löytyi valmiit Python-kirjastot, joiden avulla voitiin ohjata joko yksittäisiä releitä tai vaihtoehtoisesti kytkeä kaikki releet päälle tai pois. Testivaiheessa ei tullut kertaakaan tilannetta, jossa molemmat vakionopeudensäädintä ohjaavat releet olisivat olleet vahingossa päällä samaan aikaan.

Yksi merkittävä ongelma vakionopeudensäätimen ohjauksessa relekortin avulla havaittiin, kun ohjelma käynnistettiin auton ollessa paikallaan ja vakionopeudensäätimen kytkin auton vilkkukahvassa oli off-asennossa. Tällöin, kun vakionopeudensäädintä yritettiin kytkeä aktiiviseksi auton kulkiessa haluttua nopeutta, se ei toiminut ollenkaan edes auton omista kytkimistä. Sama ongelma havaittiin myös, mikäli jouduttiin painamaan jarrupoljinta samaan aikaan, kun ohjelma oli käynnissä. Pääteltiin, että moottorinohjausyksikköön oli ohjelmoitu suojamekanismi, joka kytkee vakionopeudensäätimen pois käytöstä, mikäli kuljettajan toiminnoissa havaitaan ristiriita. Edellä ensin mainitussa tilanteessa

RasPi yritti käyttää auton vakionopeudensäätimen kytkimiä sen ollessa pois päältä, minkä ei auton logiikan mukaan pitäisi olla mahdollista.

Edellä mainitusta vikatilasta poistuminen vaati aina auton sammuttamisen ja uudelleenkäynnistyksen. Ohjelma saatiin toimimaan vain, jos se käynnistettiin vasta, kun ajettiin sopivaa nopeutta ja auton oma vakionopeudensäädin oli kytketty aktiiviseksi painamalla set-kytkintä. Tämän jälkeen ohjelma toimi ja ohjasi vakionopeudensäädintä niin kuin oli tarkoitus. Ratkaisuksi tähän ongelmaan ohjelman käyttöliittymään lisättiin off-painike, joka ohjelman käynnistysvaiheessa on oletuksena valittuna. Adaptiivisuutta käyttääkseen kuljettajan tuli valita painikkeista haluttu ajonopeus ja etäisyys edellä ajavaan ajoneuvoon.

Auton vakionopeudensäätimen ohjaamiseen relekortin avulla kirjoitettiin ohjelma, jossa näytöllä olevista painikkeista pystyi valitsemaan halutun ajonopeuden. Kuljettajan valittua tietyn nopeuden, ohjelma luki auton CAN-väylältä ajonopeuden ja käytti vakionopeudensäätimen painikkeita muuttaakseen nopeuden vastaamaan valittua nopeutta. Tässä vaiheessa tutka ei ollut käytössä ja vakionopeudensäädin toimi perinteisen vakionopeudensäätimen tapaan. Tätä testatessa huomattiin, että vakionopeudensäätimen kytkimiä täytyy painaa vähintään 0,5 sekuntia, jotta komento välittyy moottorinohjausyksikölle. Käytännössä ohjelmaan kirjoitettiin varmuuden vuoksi 1 sekunnin viive, kun vakionopeudensäätimen kytkimiä käytetään.

Ohjelma kirjoitettiin toimimaan siten, että se kiihdyttää auton nopeutta, jos nopeus on alle valitun nopeuden ja vastaavasti hidastaa, jos auton nopeus on yli valitun nopeuden. Tässä havaittiin, että koska ohjelma lopettaa kiihdytyksen vasta, kun valittu nopeus on saavutettu, auton nopeus kasvaa sen jälkeen vielä 1–2 km/h. Koska nopeus oli kasvanut yli valitun nopeuden, ohjelma alkoi hidastaa auton nopeutta. Tällöin nopeus taas laski alle valitun, jolloin ohjelma alkoi taas kiihdyttää. Ratkaisuksi ohjelmaan tehtiin lisäys, joka sallii nopeuden muutoksen ±2 km/h valitusta nopeudesta ilman, että se johtaa minkäänlaisiin toimenpiteisiin. Havaittiin, että tämä rauhoitti vakionopeudensäätimen käyttöä huomattavasti ja käytännössä ajonopeus pysyi useimmiten ±1 km/h valitusta.

6.3 CAN-väyläkommunikaatio

CAN-väyläkommunikaatio onnistui erinomaisesti MKR CAN Shieldin avulla. Sen ohjaus Python-ohjelmointikielellä osoittautui hyvin helpoksi. Ainoa ongelma oli, että sen virransyöttö ei saanut olla kytkettynä RasPin GPIO-nastoihin RasPia käynnistettäessä. Mikäli MKR CAN Shieldin 5 V:n ja 3,3 V:n nastat olivat kytkettyinä RasPin GPIO-pinneihin, kun RasPiin kytkettiin virta, se ei käynnistynyt, vaan näytössä näkyi pelkkää mustaa. RasPin käynnistyminen jatkui normaalisti, kun MKR CAN Shieldin virtajohtimet kytkettiin irti RasPista. Ongelma ratkaistiin jatkossa siten, että MKR CAN Shieldin virransyöttö ohjattiin relekortin kautta ja ohjelmaan kirjoitettiin komento, joka kytkee siihen virran päälle vasta, kun ohjelma käynnistyy.

6.4 Järjestelmän testaus käytännössä

Yksi merkittävä ongelma järjestelmää testatessa oli mutkat. TF03:n lähettämä valokeila on niin kapea, että hyvin usein edessä oleva auto poistui mutkassa sen mittausalueelta, jolloin ohjelma alkoi kiihdyttää auton vauhtia. Esimerkiksi Kehä I:llä ajaessa järjestelmä toimi pääasiassa täysin tarkoituksenmukaisesti, mutta esimerkiksi Lintuvaaran ja Leppävaaran liittymien välissä mutkat ovat niin jyrkät, että tutka kadotti edessä ajavan ajoneuvon, vaikka ajettiin lyhyimmällä välimatka-asetuksella. Ratkaisuksi tähän olisi hankkia 2D- tai 3D-laserskanneri tutkan tilalle, jolloin saadaan mitattua etäisyyksiä laajemmalta alueelta, mutta tämän projektin puitteissa se ei ollut mahdollista.

Nopeudenvalintapainikkeet toimivat käytännössä moitteettomasti varsinkin sopivan hystereesin löydyttyä. Ne tuntuivat käytössä hyvin käytännölliseltä vaihtoehdolta ja myös jokseenkin turvallisemmilta kuin nopeuden säätö plus- ja miinuspainikkeilla. Esimerkiksi nopeusrajoituksen muuttuessa voi vain painaa näytöltä vastaavaa nopeutta ja tietokone sääti nopeuden automaattisesti. Perinteisellä vakionopeudensäätimellä tulee painaa joko plus- tai miinuspainiketta pohjassa, kunnes haluttu nopeus on saavutettu. Tämän toiminnon vaatima keskittyminen nopeusmittarin seurantaan voitiin ohjelman ansiosta suunnata muun liikenteen havainnointiin.

Ajonopeus- ja etäisyysnäytöt sekä vakionopeudensäätimen käytön ilmaisimet toimivat hyvin eikä niissä ilmennyt mitään ongelmia. Teraranger-tutkan ominaisuuksista johtuvat synkronointiongelmat katosivat täysin siirryttäessä Benewaken tutkaan ja näyttöjen lukemat päivittyivät reaaliajassa. Myöskään ohjelman suorituksessa kokonaisuudessaan ei havaittu minkäänlaisia katkoksia.

7 Yhteenveto

Projektin tavoitteet toteutuivat onnistuneesti. Yksi tavoitteista oli rakentaa omaan autoon adaptiivinen vakionopeudensäädin, ja tässä onnistuttiin suunnitelmien mukaisesti. Toinen tavoitteista oli syventyä ajoneuvon elektroniikkaan ja väylätekniikkaan sekä adaptiivisen vakionopeudensäätimen vaatimuksiin. Projektin alkuvaiheessa jouduttiin etsimään autosta CAN-väylä ja tutustumaan sen ominaisuuksiin ja viestien sisältöön. Alun perin tarkoituksena oli myös ohjata vakionopeudensäädintä väylän kautta, mutta sen viestien syklaavien tavujen takia jouduttiin etsimään keino ohjata vakionopeudensäädintä sen painikkeiden kautta. Tutkan toiminnan asettamat rajoitteet auttoivat ymmärtämään paremmin adaptiivisen vakionopeudensäätimen toimintaa ja sen vaatimuksia. Näin ollen myös toinen asetetuista tavoitteista täyttyi onnistuneesti.

Erityisen paljon haasteita toivat tutkat ja niiden ominaisuuksista johtuvat rajoitteet. Ensimmäisen tutkan synkronointiongelmat ratkesivat vaihtamalla tutka toiseen malliin, mutta se toi mukanaan uusia rajoitteita. Kylmyydestä johtuva tutkan kantaman huomattava lyheneminen olisi saatu korjattua rakentamalla tutkalle lämmitin, mutta sen kehitystä ei koettu tarpeelliseksi tässä projektissa. Tutka irrotettiin testien välissä autosta ja tuotiin sisätiloihin lämpenemään, minkä jälkeen se toimi normaalisti muutaman tunnin. Tuossa ajassa järjestelmää saatiin testattua tarpeeksi eikä lämmittimen rakennusta näin ollen pidetty tarpeellisena. Tutkan valokeilan kapeudesta johtuvat ongelmat taas olisivat vaatineet tutkan vaihtamisen mahdollisesti ajoneuvoteollisuudessakin käytettävään malliin, joka olisi ollut tähän projektiin liian hintava.

Yhteenvetona voidaan todeta, että adaptiivisen vakionopeudensäätimen rakennus omaan autoon on mahdollista melko edullisestikin, mutta sen toiminta on hyvin paljon

rajoitetumpaa kuin valmiissa markkinoilla olevissa järjestelmissä. Yksi vaihtoehto järjestelmän toimivuuden ja luotettavuuden lisäämiseksi olisi lisätä tutkien määrää ja ottaa avuksi kamera, jonka kuvasta tunnistetaan ajoneuvojen sijainti, mutta nämä toimenpiteet lisäävät huomattavasti projektissa tarvittavien osien hintaa sekä ohjelmointiin käytettävää aikaa.

Lähteet

Mercedes-Benz History of Innovation. 2019. Verkkoaineisto. Mercedes-Benz USA, LLC. <https://www.mbusa.com/mercedes/benz/innovation>. Luettu 24.9.2019.

Oil Embargo, 1973–1974. Verkkoaineisto. Office of the Historian. <https://history.state.gov/milestones/1969-1976/oil-embargo>. Luettu 22.9.2019.

The history of adaptive cruise control. 8.8.2018. Verkkoaineisto. Autonomous Vehicle International. <https://www.autonomousvehicleinternational.com/features/adas-3.html>. Luettu 24.9.2019.

The Sightless Visionary Who Invented Cruise Control. 2018. Verkkoaineisto. Smithsonian.com. <https://www.smithsonianmag.com/innovation/sightless-visionary-who-invented-cruise-control-180968418/>. Luettu 22.9.2019.

Konsolisovelluksen lähdekoodi

Esimerkkikoodi 1. Tutkan testaukseen käytetyn konsolisovelluksen lähdekoodi

```
# serial definitions ##################################################
import serial
import io
import binascii

# Open serial communication with TF03 ##################################
# device: ttyS0, baud rate: 115200

try:
      while True:
            tf03 = serial.Serial('/dev/ttyS0', 115200)
            data = tf03.read(size=9) # read 9 bytes
            datarr = bytearray(data) # convert to bytearray

            # Calculate checksum and compare to received
            rec_check = datarr[8]
            cal_check = datarr[0] + datarr[1] + datarr[2] + datarr[3]
            while cal_check > 256:
                  cal_check -= 256

            # Print data only if checksum is correct
            if cal_check == rec_check:
                  dist = datarr[2] + (256 * datarr[3])
                  dist_m = dist / 100
                  print("Distance:\t{} m".format(dist_m))
            else:
                  print("Distance:\t----- m")
except KeyboardInterrupt:
      # On KeyboardInterrupt close serial port before exiting
      tf03.close()
      print("Port closed")
      input("Press Enter to quit")
```

Graafisen käyttöliittymän lähdekoodi

Esimerkkikoodi 2. Projektissa käytetyn Python-ohjelman lähdekoodi

```
#########################################################################
# Adaptiivinen vakionopeudensäädin
# (c) Topi Orpana 2019
# Opinnäytetyö
# Metropolia Ammattikorkeakoulu
#
# Adaptive cruise control
# (c) Topi Orpana 2019
# Thesis
# Metropolia University of Applied Sciences
#########################################################################

# Logging module ########################################################
import logging
# Configure logger to log everything to a file
logging.basicConfig(level=logging.INFO,
     filename='logs/tf03log.log', filemode='w',
     format='%(asctime)s - %(name)s - %(levelname)s - %(message)s')
# Examples of usage:
# logging.debug('This is a debug message')
# logging.info('This is an info message')
# logging.warning('This is a warning message')
# logging.error('This is an error message')
# logging.critical('This is a critical message')
logging.info("\n\nLogging started")
#------------------------------------------------------------------------

# Readiness indicators ##################################################
mkr = False          # MKR CAN Shield
relay = False        # SeeedStudio Relay Board
tf = False           # Benewake TF03

# Multiprocessing definitions ###########################################
from multiprocessing import *
#------------------------------------------------------------------------

# TF03 serial definitions ###############################################
import serial
import io
import binascii
#------------------------------------------------------------------------

# SeeedStudio Relay Board definitions ###################################
from relay_lib_seeed import *
# relay_on(int_value)                   - Turns a single relay on
# relay_off(int_value)                  - Turns a single relay off
# relay_get_port_status(int_value)      - Check relay state (True = on)
# relay_all_on()                        - Turns all of the relays on
# relay_all_off()                       - Turns all of the relays off

# Set cruise control relay numbers
acc_relay = 2 # Define acceleration relay number
dec_relay = 1 # Define deceleration relay number

# Set MKR CAN Shield power relays
```

```
mkr_5v = 4      # MKR CAN Shield 5 V power input
mkr_3v3 = 3     # MKR CAN Shield 3.3 V power input

# Switch all relays off and power up MKR CAN Shield
try:
      relay_all_off()
      logging.info("All relays off")
      relay_on(mkr_3v3)
      relay_on(mkr_5v)
      logging.info("MKR CAN Shield on")
      relay = True    # Signal that Relay Board is connected
except:
      logging.error("Relay Board not connected")
      relay = False   # Signal that Relay Board is not connected
#----------------------------------------------------------------------

# CAN definitions ###################################################
import can
import time
import os
#----------------------------------------------------------------------

# GUI definitions ###################################################
from tkinter import *
import sys
import signal
#----------------------------------------------------------------------

# Open serial communication with TF03 ###############################
# device ttyS0, baud rate 115200
try:
      dis = serial.Serial('/dev/ttyS0', 115200)
      logging.info("TF03 connected")
      tf = True  # True = TF03 found and connected
except:
      logging.warning("TF03 not found in ttyS0")
      tf = False
if tf == False:
      try:
            logging.debug("Trying ttyS1")
            dis = serial.Serial('/dev/ttyS1', 115200)
            logging.info("TF03 not found in ttyS1")
            tf = True  # True = TF03 found and connected
      except:
            logging.error("TF03 not connected")
            tf = False # False = TF03 not connected


# Initialize CAN communication ######################################
try:
      mkr = False     # MKR CAN Shield not connected
      # shut down can0 in case it was left open before
      os.system("sudo /sbin/ip link set can0 down")
      time.sleep(0.1)

      # start up can0
      try:
            os.system("sudo ip link set can0 up type can bitrate 500000")
            time.sleep(0.1)
            mkr = True # MKR CAN Shield found
      except OSError:
            logging.error("No CAN board")
```

```
            mkr = False
      if mkr == True:
            try:
                  bus = can.interface.Bus(channel='can0',
                                                bustype='socketcan_native')
                  logging.info('CAN Ready')
            except OSError:
                  logging.error('Cannot find CAN board.')
                  mkr = False
except:
      logging.error("MKR CAN Shield not connected")
      mkr = False
#----------------------------------------------------------------------

# Process for CAN communication ########################################
def CAN():
      global mkr
      if mkr:
            while True:
                  bus.flush_tx_buffer()
                  logging.debug("CAN TX buffer flushed")
                  waiting = True  # waiting for correct CAN ID
                  while waiting:
                        message = bus.recv() # Wait until a message is received.
                        # Filter message
                        # hex id 320 (800 dec) contains speed data
                        if message.arbitration_id == 800:
                              logging.debug("CAN speed message received")
                              # extract data bytes from message
                              speed_data = message.data
                              waiting = False # correct data received
                        else:
                              continue

                  # speed data is located in 7th byte
                  hex_speed = speed_data[6]
                  logging.debug("CAN speed data extracted")

                  # convert to decimal
                  # and multiply by 1.176 to correct speed
                  speed_kmh = float(hex_speed) * 1.176

                  # convert to m/s
                  speed_ms = speed_kmh / 3.6
                  logging.debug("CAN speed data converted")

                  # Put speed to queue if queue not full
                  if cq.full():
                        logging.warning("CAN speed queue full")
                  else:
                        cq.put(speed_kmh)
                        logging.debug("CAN speed put to queue")
      else:
            while True:
                  if cq.full():
                        logging.warning("CAN speed queue full")
                  else:
                        cq.put(0)
                        logging.debug("Speed 0 put to queue")
                  time.sleep(0.1)
#----------------------------------------------------------------------
```

```
# Process for TF03 communication ########################################
def tf03():
      global tf       # TF03 readiness indicator
      n = 0           # Median filter index
      rdy = False           # False = filter not ready yet
      filtered = 0.0  # Median filtered measurement
      dist_m = 0.0    # Distance measurement from TF03
      old = 0.0       # Previous measurement from TF03
      new = 0.0       # New measurement from TF03
      checksum = 0    # Checksum for received data

      if tf:
            while True:
                  data = dis.read(size=9) # read 9 bytes
                  logging.debug("TF03 data read")
                  datarr = bytearray(data) # convert to bytearray
                  logging.debug("TF03 data converted to bytearray")

                  datex = binascii.hexlify(bytearray(datarr))
                  logging.debug("TF03 bytearray converted to hex")
                  logging.debug("TF03 hex data: {}".format(datex))

                  dist_cm = datarr[2] + (256 * datarr[3])
                  logging.debug("TF03 dist_cm: {}".format(dist_cm))

                  checksum = datarr[8]
                  logging.debug("TF03 checksum: {}".format(checksum))

                  checks = datarr[0] + datarr[1] + datarr[2] + datarr[3]
                  while checks > 256:
                        checks -= 256
                  logging.debug("TF03 calculated checksum: {}".format(checks))

                  if checks != checksum:
                        dist_cm = old
                        logging.error("TF03 checksum error")
                  elif dist_cm < 30:
                        dist_cm = old
                        logging.error("TF03 dist_cm < 30")

                  # Low pass filter measurements
                  old = filtered
                  new = float(dist_cm)
                  logging.debug("TF03 old: {}, new: {}".format(old,new))
                  filtered = (0.9*old + 0.1*new)
                  logging.debug("TF03 filtered: {}".format(filtered))

                  dist_m = filtered / 100
                  logging.debug("TF03 dist_m: {}".format(dist_m))

                  #Put distance in queue if queue not full
                  if tq.full():
                        logging.warning("TF03 queue full")
                  else:
                        tq.put(dist_m)
                        logging.debug("Distance put to queue")
      else:
            while True:
                  if tq.full():
                        logging.warning("TF03 queue full")
                  else:
                        tq.put(0)
```

```
                    logging.debug("Distance 0 m put to queue")
               time.sleep(0.1)
#---------------------------------------------------------------------

# Process for calculations ###################################################
def calculations():
     while True:
          speed_kmh = cq.get() # get speed from queue
          speed_ms = speed_kmh / 3.6 # convert to m/s

          dist_m = tq.get() # get distance from queue

          if speed_kmh == 0: # case vehicle stopped
               dist_s = 999 # distance in seconds = 999
          else:
               dist_s = dist_m / speed_ms # calculate distance in seconds

          logging.debug("Distance (s) calculated")

          # Put calculation in queue if queue not full
          if sq.full():
               logging.warning("Calculations queue full")
          else:
               sq.put(dist_s)
               logging.debug("Calculations put to queue")
#---------------------------------------------------------------------

# Catch distance selection from radio buttons ###########################
def Dis_selection():
     selection = Dis.get()
     sel_s = float(selection) / 10 # Divide by 10 to get seconds
     logging.info("Selected distance: ", sel_s, " s")
     rq.put(selection) # Put selection in queue
     logging.debug("Distance selection put to queue")
#---------------------------------------------------------------------

# Catch speed selection from radio buttons ##############################
def Spd_selection():
     selection = Spd.get()
     logging.info("Selected speed: ", selection, " km/h")
     ssq.put(selection) # Put selection in queue
     logging.debug("Speed selection put to queue")
#---------------------------------------------------------------------

# Process to control relays that send commands to cruise control #######
def relay_control():
     global relay

     rel = 0              # Variable for distance setting
     set_kmh = 0          # Variable for speed setting

     if relay:
          while True:
               # Get speed from queue
               speed_km = cq.get()

               # If distance selection not changed
               # keep last selected
               if rq.empty():
                    rel = rel
               else:
                    rel = rq.get() # Get selection from queue
```

```
                    rel_s = float(rel) / 10
                    dist_s = sq.get() # Get actual distance to ahead object

                    # Get speed selection from queue
                    if ssq.empty():
                         set_kmh = set_kmh
                         logging.debug("Set speed queue empty, using same")
                    else:
                         set_kmh = ssq.get()
                         logging.debug("Set speed fetched from queue")

                    # Use cruise control only if a distance is selected
                    # and speed is above 35 km/h
                    if rel == 0:
                         logging.info("ACC off")
                         relay_off(acc_relay)
                         relay_off(dec_relay)
                    else:
                         if speed_km > 35:
                              # Slow down if ahead object too close
                              # or if speed too high
                              if (dist_s < rel_s or
                               speed_km > (set_kmh + 2)):
                                   logging.info("Decelerating")
                                   relay_off(acc_relay)
                                   relay_on(dec_relay)
                                   time.sleep(1)

                              # Accelerate if ahead object far enough
                              # and own speed below set speed
                              elif (dist_s > (rel_s + 0.2) and
                               speed_km < (set_kmh - 2)):
                                   logging.info("Accelerating")
                                   relay_off(dec_relay)
                                   relay_on(acc_relay)
                                   time.sleep(1)

                              # Maintain same speed if distance to
                              # ahead object same as set distance
                              else:
                                   logging.info("Cruising")
                                   relay_off(acc_relay)
                                   relay_off(dec_relay)

                         else:
                              logging.warning("Speed under 35 km/h")
                              relay_off(acc_relay)
                              relay_off(dec_relay)
          else:
               logging.error("Relay board not connected")
#------------------------------------------------------------------------

# Catch exit button press ##################################################
def exit_button_pressed():
     logging.info("Exiting program")
     root.destroy() # Destroy GUI
     C.terminate() # Terminate CAN process
     T.terminate() # Terminate tf03 process
     Ca.terminate() # Terminate Calculations process
     R.terminate() # Terminate Relay control process
     relay_all_off() # Turn all relays off before exiting
     os._exit(0)
```

```
#-------------------------------------------------------------------

# GUI definitions ######################################################
class GUI:
     def __init__(self, master):
          logging.debug("Initializing GUI")

          self.master = master
          master.title("Adaptive Cruise Control")

          # Set window to fullscreen (exit fullscreen by pressing ESC)
          logging.debug("Setting fullscreen mode")
          master.attributes("-fullscreen", True)
          master.bind("<Escape>", self.end_fullscreen)

          # Define text display for distance (meters) info
          logging.debug("Creating label for distance (m)")
          self.dist_var = StringVar()
          self.label_d = Label(master, textvariable=self.dist_var,
                               font="Piboto 20", padx=10, width=25, height=3)
          self.label_d.grid(row=0, column=0, columnspan=2)
          self.dist_info = "Distance: " + "{:.2f}".format(dist_m) + " m"
          self.dist_var.set(self.dist_info)

          # Define text display for distance (seconds) info
          logging.debug("Creating label for distance (s)")
          self.dist_s_var = StringVar()
          self.label_d_s = Label(master, textvariable=self.dist_s_var,
                               font="Piboto 20", padx=10, width=25, height=3)
          self.label_d_s.grid(row=0, column=2, columnspan=2)
          self.dist_s_info = "Distance: " + "{:.2f}".format(dist_s) + " s"
          self.dist_s_var.set(self.dist_s_info)

          # Define text display for speed info
          logging.debug("Creating label for speed (km/h)")
          self.speed_var = StringVar()
          self.label_s = Label(master, textvariable=self.speed_var,
                               font="Piboto 20", padx=10, width=25, height=3)
          self.label_s.grid(row=1, column=2, columnspan=2)
          self.speed_info = ("Speed: "
                             + "{:.0f}".format(speed_kmh) + " km/h")
          self.speed_var.set(self.speed_info)

          # Define radio buttons for distance selection
          logging.debug("Creating radio buttons for distance selection")
          R10 = Radiobutton(master, text="1.0 s", indicatoron=False,
                                    variable=Dis, value=10,
                                    command=Dis_selection,
                                    width=20, height=2)
          R10.grid(row=2, column=0)
          R15 = Radiobutton(master, text="1.5 s", indicatoron=False,
                                    variable=Dis, value=15,
                                    command=Dis_selection,
                                    width=20, height=2)
          R15.grid(row=2, column=1)
          R20 = Radiobutton(master, text="2.0 s", indicatoron=False,
                                    variable=Dis, value=20,
                                    command=Dis_selection,
                                    width=20, height=2)
          R20.grid(row=2, column=2)
          R0 = Radiobutton(master, text="OFF", indicatoron=False,
                                    variable=Dis, value=0,
```

```
                                    command=Dis_selection,
                                    width=20, height=2)
R0.grid(row=2, column=3)

#Define radio buttons for speed selection
logging.debug("Creating radio buttons for speed selection")
R40 = Radiobutton(master, text="40 km/h", indicatoron=False,
                                    variable=Spd, value=40,
                                    command=Spd_selection,
                                    width=20, height=2)
R40.grid(row=3, column=0)
R50 = Radiobutton(master, text="50 km/h", indicatoron=False,
                                    variable=Spd, value=50,
                                    command=Spd_selection,
                                    width=20, height=2)
R50.grid(row=3, column=1)
R60 = Radiobutton(master, text="60 km/h", indicatoron=False,
                                    variable=Spd, value=60,
                                    command=Spd_selection,
                                    width=20, height=2)
R60.grid(row=3, column=2)
R70 = Radiobutton(master, text="70 km/h", indicatoron=False,
                                    variable=Spd, value=70,
                                    command=Spd_selection,
                                    width=20, height=2)
R70.grid(row=3, column=3)
R80 = Radiobutton(master, text="80 km/h", indicatoron=False,
                                    variable=Spd, value=80,
                                    command=Spd_selection,
                                    width=20, height=2)
R80.grid(row=4, column=0)
R90 = Radiobutton(master, text="90 km/h", indicatoron=False,
                                    variable=Spd, value=90,
                                    command=Spd_selection,
                                    width=20, height=2)
R90.grid(row=4, column=1)
R100 = Radiobutton(master, text="100 km/h", indicatoron=False,
                                    variable=Spd, value=100,
                                    command=Spd_selection,
                                    width=20, height=2)
R100.grid(row=4, column=2)
R120 = Radiobutton(master, text="120 km/h", indicatoron=False,
                                    variable=Spd, value=120,
                                    command=Spd_selection,
                                    width=20, height=2)
R120.grid(row=4, column=3)

# Define application exit button
logging.debug("Creating exit button")
exit_button = Button(master, text="EXIT",
                           command=exit_button_pressed,
                           width=20, height=2)
exit_button.grid(row=5, column=0)

logging.debug("Creating CC indicators")
# Define acceleration indicator
self.acc_var = StringVar()
self.label_acc = Label(master, textvariable=self.acc_var,
                           font="OpenSymbol 20")
self.label_acc.grid(row=1, column=0)
self.acc_var.set(u"\u21e7")
```

```
            # Define deceleration indicator
            self.dec_var = StringVar()
            self.label_dec = Label(master, textvariable=self.dec_var,
                                   font="OpenSymbol 20")
            self.label_dec.grid(row=1, column=1)
            self.dec_var.set(u"\u21e9")

            logging.debug("GUI ready")

      # Process for updating label texts
      def update_labels(self):
            # Distance in meters
            self.dist_m = tq.get()
            self.dist_info = ("Distance: "
                              + "{:.2f}".format(self.dist_m) + " m")
            self.dist_var.set(self.dist_info)
            logging.debug("dist_m updated")

            #Distance in seconds
            self.dist_s = sq.get()
            self.dist_s_info = ("Distance: "
                              + "{:.2f}".format(self.dist_s) + " s")
            self.dist_s_var.set(self.dist_s_info)
            logging.debug("dist_s updated")

            # Speed in km/h
            self.speed_kmh = cq.get()
            self.speed_info = ("Speed: "
                               + "{:.0f}".format(self.speed_kmh) + " km/h")
            self.speed_var.set(self.speed_info)
            logging.debug("speed_kmh updated")

            # Acceleration and deceleration indicators
            if (relay_get_port_status(acc_relay)):
                  self.acc_var.set(u"\u21e7")
                  self.dec_var.set(" ")
            elif (relay_get_port_status(dec_relay)):
                  self.dec_var.set(u"\u21e9")
                  self.acc_var.set(" ")
            else:
                  self.acc_var.set(" ")
                  self.dec_var.set(" ")
            logging.debug("CC indicators updated")

            logging.debug("All labels updated")

      # Process for ending fullscreen
      def end_fullscreen(self, event=None):
            master.attributes("-fullscreen", False)
#---------------------------------------------------------------------------

# Main program ###########################################################
if __name__ == '__main__':

      # CAN Process
      C = Process(target=CAN) # Define C as CAN
      cq = Queue(maxsize=1) # Define queue for CAN messages
      C.start() # Start CAN process
      logging.debug("CAN process started")

      # TF03 Process
      T = Process(target=tf03) # Define T as tf03
```

```
tq = Queue(maxsize=1) # Define queue for tf03 messages
T.start() # Start tf03 process
logging.debug("TF03 process started")

# Calculations Process
Ca = Process(target=calculations) # Defince Ca as calculations
sq = Queue(maxsize=1) # Define queue for calculations
Ca.start() # Start calculations process
logging.debug("Calculations process started")

# First measurements before GUI
dist_m = tq.get() # Get first measurement
logging.debug("First tq.get() successful")
speed_kmh = cq.get() # Get first measurement
logging.debug("First cq.get() successful")
dist_s = sq.get() # Get first calculation
logging.debug("First sq.get() successful")

# Relay control Process
R = Process(target=relay_control) # Define R as relay_control
rq = Queue(maxsize=1) # Define queue for distance setting
ssq = Queue(maxsize=1) # Define queue for speed setting
tgl = Queue(maxsize=1) # Define queue for ON/OFF button toggle
R.start() # Start relay control process
logging.debug("Relay control started")

# Start Processes
root = Tk()
Dis = IntVar()
Dis.set(0)
Spd = IntVar()
gui = GUI(root)
logging.debug("GUI started")

# Loop for updating measurements and GUI
logging.debug("Entering main loop")
while True:
      gui.dist_m = tq.get() # Get next distance measurement from queue
      gui.dist_info = "Distance: " + "{:.2f}".format(dist_m) + " m"
      gui.dist_var.set(gui.dist_info)
      gui.speed_kmh = cq.get() # Get next speed measurement from queue
      gui.update_labels()
      root.update() # Update GUI
      logging.debug("GUI updated")
```