Eppu Ainola

Ohjelmiston latausaseman suunnittelu taajuusmuuttajien massatuotantoon

Sähkötekniikan korkeakoulu

Diplomityö, joka on jätetty opinnäytteenä tarkastettavaksi diplomi-insinöörin tutkintoa varten Espoossa 6.10.2013.

Työn valvoja:

Prof. Seppo Ovaska

Työn ohjaaja:

DI Jari Mäkilä

Aalto-yliopisto
Sähkötekniikan korkeakoulu

AALTO-YLIOPISTO
SÄHKÖTEKNIIKAN KORKEAKOULU

DIPLOMITYÖN
TIIVISTELMÄ

Tekijä: Eppu Ainola

Työn nimi: Ohjelmiston latausaseman suunnittelu taajuusmuuttajien massatuotantoon

Päivämäärä: 6.10.2013 Kieli: Suomi Sivumäärä: 8+92

Sähkötekniikan laitos

Professuuri: Teollisuuselektroniikka Koodi: S-81

Valvoja: Prof. Seppo Ovaska

Ohjaaja: DI Jari Mäkilä

Työssä tutustutaan Flash-muistiteknologiaan ja suunnitellaan asema, joka lataa ohjelmiston taajuusmuuttajan Flash-muistiin. Flash-muistien käsittely rajataan kahteen yleisimpään tyyppiin: NAND- ja NOR-muisteihin. Flash-muisteja tarkastellaan perusteiden lisäksi siltä kannalta, että voidaanko taajuusmuuttajissa siirtyä käyttämään NAND-tyyppistä Flash-muistia nykyisen NORin sijaan, ja kuinka suuri operaatio on elektroniikan ja vaadittavien ohjelmistojen kannalta. Latausasema suunnitellaan tukemaan usean taajuusmuuttajan yhtäaikaista ohjelmointia tuotannon tehostamiseksi. Se on tarkoitus ottaa käyttöön työn toimeksiantajan tuotantolinjalla. Työ keskittyy latausaseman elektroniikan ja ohjelmiston suunnitteluun. Mekaniikka ja laitteen käyttöönotto eivät kuulu työn rajaukseen.

Flash-muisteja tutkitaan tieteellisten julkaisujen, alan kirjallisuuden ja muistivalmistajien datalehtien sekä sovellusohjeiden perusteella. Valmistajien julkaisemaa tietoa käytetään Flash-muistien nykyaikaisten ominaisuuksien selvittämiseen. Latausasema toteutettiin käyttäen mahdollisimman paljon toimeksiantajalla olevaa teknologiaa sekä painottaen helppoa ylläpidettävyyttä.

Työn tuloksena todetaan NANDien olevan varteenotettava vaihtoehto teollisuudessa, erityisesti jos pidättäydytään NAND-muisteissa, jotka tallentavat vain yhden bitin tietoa yhteen muistisoluun, eli ovat yksitasoisia. Latausasemalla saavutetaan kuusinkertainen ohjelmiston latausnopeus samalla tehtaan pinta-alalla kuin mitä laitteen edeltäjä vaatii.

Avainsanat: Flash, NAND, NOR, Teollisuuden tuotantolaite

AALTO UNIVERSITY
SCHOOL OF ELECTRICAL ENGINEERING

ABSTRACT OF THE
MASTER'S THESIS

Author: Eppu Ainola

Title: Firmware uploading station for the use in mass production of variable-frequency drives

Date: 6.10.2013 Language: Finnish Number of pages: 8+92

Department of Electrical Engineering

Professorship: Industrial Electronics Code: S-81

Supervisor: Prof. Seppo Ovaska

Instructor: M.Sc. Jari Mäkilä

This thesis examines Flash memory technology and a firmware uploading station for variable-frequency drives is designed. Only two of the most common Flash memory types are considered: NAND and NOR. It is investigated, could NAND type Flash memories be used in industrial applications instead of NOR, and how difficult would the migration be. The device is designed to support uploading firmware to multiple variable-frequency drives simultaneously. It is intended to be used in an actual factory environment. The focus is on the design of electronics and software for the station. Mechanical and deployment aspects are not included.

Flash memories are studied from scientific articles, appropriate literature, and the datasheets and application notes published by various memory manufacturers. Information from the datasheets is useful in learning the properties of the latest Flash memory integrated circuits. The firmware uploading station is implemented by using existing technology as widely as possible and focusing on the ease of maintenance of the device.

It is concluded that NAND type Flash memories could be used in industrial applications, especially if single level cell NAND memories are used. Single level cell means that each memory cell stores only one bit of information. The uploading station achieves sixfold throughput when compared to its predecessor which has similar mechanical dimensions.

Keywords: Flash, NAND, NOR, Industrial production equipment

Esipuhe

Diplomityö osoittautui erittäin kirjaimellisesti opinnäytetyöksi, koska sen parissa pääsin näyttämään oppimiani taitoja eri tieteenaloilla. Työhön kuului niin elektroniikka- kuin ohjelmistosuunnitteluakin järjestelmätasolta hyvin matalaan tasoon asti. Lisäksi projektin myötä pääsin tutustumaan projektinhallintaan, mekaniikan määrittämiseen ja nykyaikaiseen haihtumattomaan muistiteknologiaan syvällisesti.

Työ oli varsin teollisuuteen suuntautunut ja siinä painotettiin toimeksiantaja ABB Oy:tä kiinnostavia asioita. Diplomityö sisälsi laajan käytännön osuuden, jossa suunniteltiin laite tehtaan tuotantolinjastolle. Toivon toimeksiantajayrityksen hyötyvän diplomityöstäni, koska ainakin minä hyödyin siitä uusina kokemuksina, taitoina ja tietoina.

Kiitos työn ohjaaja Jari Mäkilälle, joka tarjosi monipuolisen ja kiinnostavan diplomityöaiheen sekä antoi palautetta viikoittain työn etenemisestä. Kiitos myös työn valvojalle, professori Seppo Ovaskalle, kiinnostuksesta työhön ja erittäin hyvästä tavoitettavuudesta sekä yksityiskohtaisesta palautteesta.

Erityismainintana kiitos myös ABB Oy:n *pienjänniteosaston (LVAC, engl. Low Voltage Alternating Current)* ohjauselektroniikkatiimille rakentavasta palautteesta elektroniikkasuunnitelmistani ja Tomi Kainusalmelle yhteyshenkilönä toimimisesta minun ja ABB:n alihankkijoiden välillä.

Espoo 6.10.2013

Eppu Ainola

Sisällysluettelo

Lyhenteet

ASCII	American Standard Code for Information Intechange Amerikkalainen tiedonvaihdon standardikoodi
CFI	Common Flash Interface Yleinen Flash-rajapinta
CHE	Channel Hot Electron Kuumakanavaelektronimenetelmä
DRAM	Dynamic Random Access Memory Muuttuva satunnaisosoitusmuisti
ECC	Error Correction Code Virheenkorjauskoodi
EEPROM	Electronically Erasable Programmable Read-Only Memory Sähköisesti tyhjennettävä lukumuisti
EIA	Electronic Industries Alliance Elektroniikkateollisuuden liittouma
e·MMC	Embedded MultiMediaCard Sulautettu muistikortti
FCT	Functional Test Toiminnallinen testi
FET	Field Effect Transistor Kanavatransistori
FN	Fowler–Nordheim Fowler–Nordheim (kahden henkilön sukunimet)
FPGA	Field-Programmable Gate Array Kenttäohjelmoitava porttimatriisi
ICT	In-Circuit Test Piirin sisäinen testi
JEDEC	Joint Electron Devices Engineering Council Elektronilaitetekniikan yhteisneuvosto
LED	Light-Emitting Diode Valodiodi
LVAC	Low Voltage Alternating Current ABB Oy:n pienjännitetuotteisiin keskittyneen osaston nimi
ONFI	Open NAND Flash Interface Avoin NAND Flash -rajapinta
PCBA	Printed Circuit Board Assembly Piirilevykokonaisuus
Q	Quarter of a year Vuosineljännes
MLC	Multi-Level Cell Monitasoinen solu
RS	Recommended Standard Suositeltu standardi
SBPI	Self-Boosted Program Inhibit Itsetehostettu kirjoituksen esto
SFDP	Serial Flash Discoverable Parameters Sarjamuotoisten Flashien kyseltävät parametrit

SLC	Single-Level Cell Yksitasoinen solu
TIA	Telecommunications Industry Association Tietoliikenneteollisuuden yhdistys
UFS	Universal Flash Storage Yleinen Flash-muisti
UPS	Uninterruptable Power Supply Varavirtalähde
USB	Universal Serial Bus Yleissarjaväylä
USD	United States Dollar Yhdysvaltain dollari
XIP	eXecute-In-Place Suora-ajo

1 Johdanto

Pienjänniteluokan[1] taajuusmuuttajia valmistetaan ja myydään ympäri maailmaa. Jokainen taajuusmuuttaja tarvitsee ohjelmiston toimiakseen, ja mitä nopeammin se voidaan syöttää kohdelaitteeseen, sitä enemmän niitä voidaan valmistaa ja myydä. Elektroniikka- tai ohjelmistosuunnittelija voi vaikuttaa valitun koteloidun muistipiirin kirjoitusaikaan ainoastaan piirin ominaisuuksien määrittämään rajaan asti, mutta laskennallisesti aikaa voidaan lyhentää syöttämällä ohjelmisto useaan taajuusmuuttajaan yhtäaikaisesti. Jatkossa tätä varsinaisen ohjelmointiajan jakamista rinnakkain ohjelmoitavien taajuusmuuttajien lukumäärällä kutsutaan laskennalliseksi ohjelmointinopeudeksi.

Työn toimeksiantaja ABB Oy on kansainvälinen yritys, joka valmistaa sähkövoima- ja automaatioteknologian tuotteita. Helsingissä yritys suunnittelee erityisesti taajuusmuuttajia sekä vaihtosuuntaajia. Nyt ABB on julkaisemassa uutta pienjännitetaajuusmuuttajien mallistoa, johon tämä diplomityö liittyy. Onnistuessaan ohjelmointiaseman toivotaan tuovan taloudellista hyötyä, koska yksi työntekijä pystyy aseman avulla ohjelmoimaan taajuusmuuttajia nopeammin kuin nyt.

Ensisijaisena tutkimuskysymyksenä selvitetään, voidaanko taajuusmuuttajien massavalmistusta nopeuttaa tarkoitukseen suunniteltavalla ohjelmiston latausasemalla? Käytännön osuudessa suunnitellaan ja toteutetaan tämä latausasema, jonka ydintoiminto on ohjelmiston syöttäminen useaan taajuusmuuttajaan yhtäaikaisesti. Yhdellä asemalla on tarkoitus korvata useita yhden taajuusmuuttajan kerrallaan ohjelmoivia laitteita.

Tutkimus rajataan tarvittavan elektroniikan ja ohjelmistojen suunnitteluun. ABB:n ohjelmistotiimi on suunnitellut komentorivisovelluksen, joka syöttää ohjelmiston yhteen taajuusmuuttajaan. Latausasema käyttää tätä olemassa olevaa sovellusta yhtenä komponenttina kokonaisuudessa. Mekaniikkasuunnittelu päätettiin jättää alihankkijalle, jolla on asiantuntemusta samankaltaisista laitteista. Mekaniikalle laaditaan kuitenkin vaatimukset ja rajat, jotta alihankkija tietää, mitä siltä odotetaan. Käyttöönotto jää myös työn ulkopuolelle diplomityön rajallisen tavoitesuoritusajan vuoksi.

Ohjelmiston latauksen tehostaminen erityisellä latausasema on onnistuessaan toimiva ratkaisu, mutta se ei ota kantaa juurisyyhyn eli käytetyn NOR-tyyppisen Flash-muistin[2] hitaaseen kirjoitusnopeuteen. Toissijaisina tutkimuskysymyksinä selvitetään, mistä ohjelmoinnin hitaus johtuu ja onko mahdollista siirtyä tulevaisuudessa käyttämään NAND-tyyppistä Flash-muistia, jonka kirjoitusnopeus on moninkertainen, mutta luotettavuus pitkäikäisessä sovelluksessa on työn toimeksiantajalle epäselvä. Jos päädytään siihen, että NAND-muistit ovat varteenotettava vaihtoehto, luonteva jatkokysymys on, miten suuri työ muutoksen tekeminen on elektroniikan ja ohjelmiston kannalta?

Flash-muisteihin liittyvä tarkastelu rajataan ainoastaan NAND- ja NOR-tyyppisiin muisteihin, koska muiden Flash-tyyppien saatavuus on työn kirjoitushetkellä näihin verrattuna heikko. Tietolähteinä käytetään pääosin tieteellisiä julkaisuja sekä alan kirjallisuutta, joka pohjautuu vahvasti samoihin julkaisuihin. Nopeasti kehittyvällä alalla tutkimukset sisältävät kuitenkin usein joko aavistuksen vanhentunutta tietoa tai niin tuoret-

[1] Euroopan unionissa pienjännite tarkoittaa 50–1000 V vaihtovirtaa tai 75–1500 V tasavirtaa [1].

[2] Flash-muisti sekä sen NAND- ja NOR-tyypit ovat termeinä vakiintuneita, joten niille ei esitetä suomennoksia.

ta tietoa, ettei siihen liittyviä sovelluksia vielä ole. Muistien ajantasaisten ominaisuuksien selvittämisessä turvaudutaan muistipiirien datalehtiin ja muistivalmistajien sovellusohjeisiin. Tutkimuskysymyksiä tarkastellaan ABB:n elektroniikkasuunnittelijan näkökulmasta. Integroidun muistipiirin sisus ei kuulu tarkasteluun toimintaperiaatetta tarkemmin, koska piirit ostetaan niihin erikoistuneelta valmistajalta. Toisaalta myöskään tietokoneiden massamuisteina käytettyjä puolijohdekiintolevyjä ei käsitellä, koska taajuusmuuttajien ohjelmisto ei tarvitse niin paljon säilytystilaa.

Työssä käsitellään aluksi taajuusmuuttajan ja Flash-muistin perusteet siinä laajuudessa kuin ne ovat työn kannalta oleellisia. Sen jälkeen tutustutaan Flash-muisteihin liittyvään teoriaan, joka on oleellista toissijaisiin tutkimuskysymyksiin vastattaessa. Vielä ennen käytännön latausaseman suunnitelmia, kuvaillaan aseman tuleva käyttöympäristö sekä asetetaan sille vaatimukset, jotka aseman tulee täyttää. Lopuksi esitetään teoriasta nousseita suosituksia sekä tarkistetaan latausaseman toimivuus vaatimuksiin nähden tutkimustuloksissa ja kootaan työ yhteenvedossa.

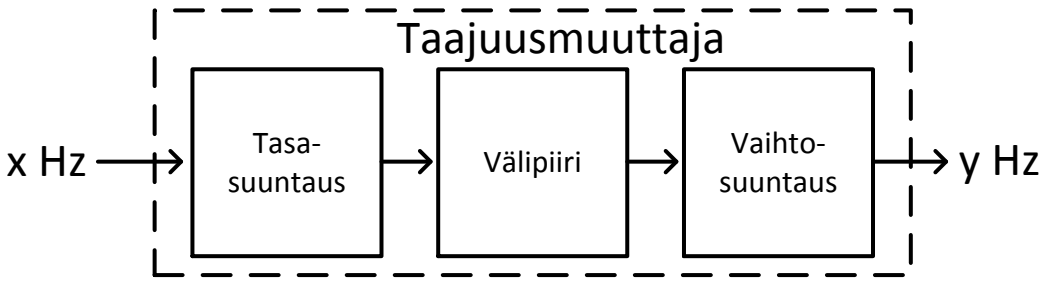
2 Taajuusmuuttaja

Työssä suunniteltava ohjelmiston latausasema syöttää ohjelmiston taajuusmuuttajien muistipiireille. Tässä luvussa selitetään, mikä on taajuusmuuttaja? Kovin tarkasti taajuusmuuttajiin ei perehdytä, koska työssä käsitellään niiden ydintoiminnallisuuden sijaan enemmän taajuusmuuttajan ohjauspiirikortin rakenteellisia ominaisuuksia. Taajuusmuuttajan ohjauskortin rakenne muistuttaa elektroniikan osalta muuta nykyaikaista digitaalista teollisuuselektroniikkaa.

Taajuusmuuttajia on useita erikokoisia, suurimmat huoneen kokoisia, mutta tässä käsitteellä tarkoitetaan pienjännitetaajuusmuuttajaa. Niiden koko vaihtelee paljon tehoarvon mukaan, mutta yleistäen kaikki ovat helposti siirrettävän kokoisia. Tyypillisenä esimerkkinä pienjännitetaajuusmuuttajasta voidaan mainita ABB:n ACS550-sarjan taajuusmuuttaja, jonka pienin 1,1 kW versio on kooltaan 369 x 125 x 212 mm (korkeus, leveys, syvyys) ja suurin 160 kW versio on 888 x 302 x 400 mm (korkeus, leveys, syvyys) [2].

2.1 Ydintoiminto

Taajuusmuuttajan toimintaperiaate voidaan selittää kolmella lohkolla, kuten kuvassa 1 on esitetty. Tasasuuntaaja ja välipiiri eivät ole välttämättömiä, jolloin puhutaan suorasta taajuusmuuttajasta. Siinä tapauksessa sisään tuleva vaihtosähkö muutetaan toisenlaiseksi ainoastaan puolijohdekytkimillä. Työn kannalta olennaisempi on kuitenkin välipiirillinen malli, joka sisältää kaikki kolme kuvan 1 lohkoa. Jännitevälipiirillisen taajuusmuuttajan tapauksessa tasasuuntaaja muuttaa sisään tulevan vaihtojännitteen tasajännitteeksi, joka varastoidaan välipiiriin. Vaihtosuuntaaja puolestaan sisältää ohjattavia puolijohdinkytkimiä, joilla tasajännite pilkotaan halutunlaiseksi.



Kuva 1: Yleistetty taajuusmuuttajan periaatekuva. Jos taajuusmuuttaja sisältää kaikki kuvan osat, sitä kutsutaan välipiirilliseksi taajuusmuuttajaksi.

Puolijohdekytkimillä on vain kaksi tilaa – johtava ja estävä –, mutta oikealla kytkemistavalla voidaan muodostaa siniaalto, jonka taajuus on teoriassa vapaasti valittavissa ja amplitudi on korkeintaan yhtä suuri kuin puolet välipiirin jännitteestä. Niiranen [3, s. 223–236] selittää, että kun välipiirin jännitettä katkotaan eri taajuisiksi tai pituisiksi pulsseiksi, kuorman induktanssi suodattaa pulssijonon mielivaltaiseksi jännitteeksi välipiirin asettamalla jännitealueella. Muuttamalla edelleen pulssien pituutta tai kytkemistaajuutta, jännite muuttuu ja voidaan muodostaa halutun taajuinen ja

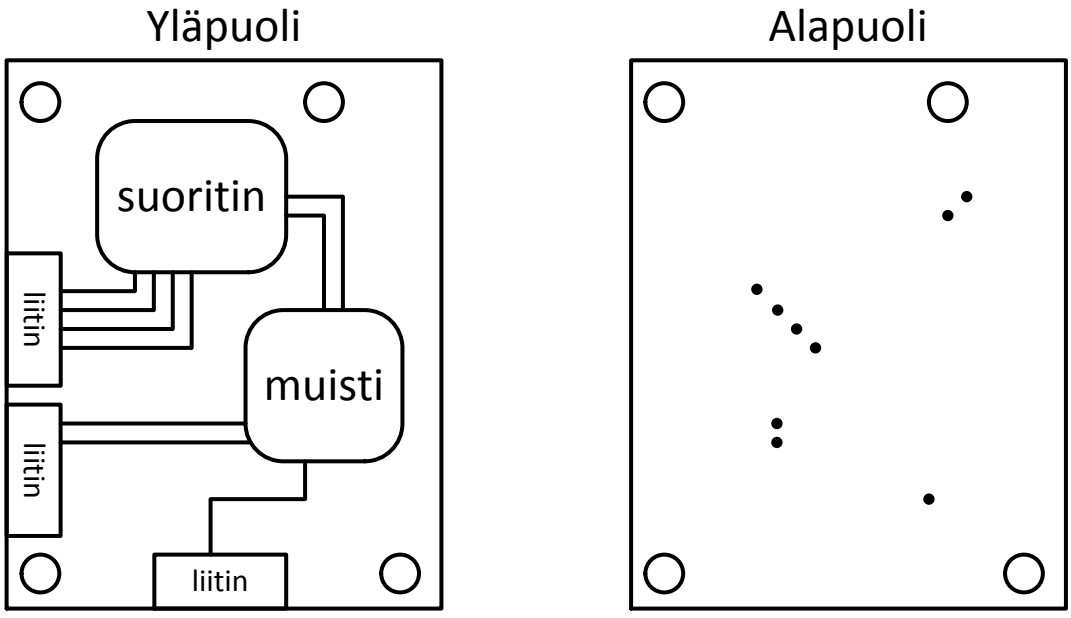
amplitudinen vaihtojännite. Jos kuorma ei ole induktiivinen, pulssijono tulee suodattaa puhtaamman aaltomuodon saamiseksi.

Taajuutta ja aallon amplitudia muuttamalla voidaan vaihdella esimerkiksi moottorin pyörimisnopeutta tai vääntömomenttia. Moottorin pyöriminen voidaan valjastaa hyötytyöhön, esimerkiksi pumppuna tai puhaltimena. Muuttajan – jonka yksi erikoistapaus taajuusmuuttaja on –, moottorin sekä näiden säädön muodostamaa kokonaisuutta kutsutaan sähkökäytöksi [3, s. 13].

2.2 Rakenne

Jäljemmän tekstin kannalta taajuusmuuttajan rakenne on sen toiminnallisuutta tärkeämpi. Se sisältää edellä mainittujen tehoelektroniikan komponenttien lisäksi muun muassa ohjauselektroniikkaa. Ohjauselektroniikalla tarkoitetaan tässä tekstissä pääsuorittimen sekä päämuistin sisältävää *piirilevykokonaisuutta (engl. Printed Circuit Board Assembly, PCBA)*.

Ohjauselektroniikka voi olla erilainen eri taajuusmuuttajissa. Yksi mahdollisuus, johon tässä työssä keskitytään, on integroidulla mikropiirillä oleva pääsuoritin sekä Flash-tyyppinen päämuisti, jossa suoritettava ohjelmisto sijaitsee. Kuvassa 2 tätä on havainnollistettu. Näiden lisäksi taajuusmuuttajassa voi olla apusuorittimia ja apumuisteja. Tässä esimerkissä ohjauselektroniikka sijaitsee omalla piirilevyllään, jossa on lisäksi muun muassa sähkömekaanisia liittimiä sekä testipisteitä massatuotannon automatisoitua testausta varten.



Kuva 2: Ohjauselektroniikan piirikortin havainnekuva. Oikealla olevan kuvan pisteet ovat testipisteitä, joiden kautta johtimiin voidaan kytkeytyä neuloilla tuotantolinjan testausvaiheissa.

Erityisesti ohjauselektroniikka muistuttaa monia muita digitaalisesti ohjattuja elektroniikkatuotteita. Esimerkiksi hissi tai teollisuusrobotti voi sisältää ohjauselektroniikan, jo-

ka käyttää samoja komponentteja ja jonka rakenne on samankaltainen. Sovelluskohtainen ohjelmisto on erilainen eri sovelluksissa.

Ohjelmisto voidaan syöttää muistipiirille muutamilla eri tavoilla. Esimerkiksi ohjelmoiva laite voi kommunikoida suoraan muistipiirin kanssa tai suorittimen läpi siten, että suoritin kirjoittaa tiedon muistille. Jälkimmäinen vaihtoehto vaatii, että ohjauskorttiin on jo ladattu *alkulatausohjelma (engl. bootloader)*. Osa mikrokontrolleripiireistä sisältää Flash-muistia erityisesti alkulatausohjelmaa varten. Atmelin sovellusohje [4] perustelee ohjelmiston kirjoittamista kahdessa osassa – mikrokontrolleriin alkulatausohjelma ja päämuistiin pääohjelma. Jos mikrokontrolleripiiri tukee alkulatausohjelman salausta, taajuusmuuttajavalmistajan sovellusohjelman kopiointi tai *takaisinmallinnus (engl. reverse engineering)* on tehokkaasti estetty, vaikka kilpailija pääsisi käsiksi pääohjelman levykuvaan esimerkiksi uuden ohjelmistopäivityksen myötä. Salausmenetelmä toimii siten, että mikrokontrolleri estää muuttumattoman alkulatausohjelman lukemisen ilman oikeaa avainta. Näin ollen pääohjelma kaikkine päivityksineen voidaan antaa asiakkaan päivitettäväksi salattuna ja salaus puretaan vain ajettaessa. Kaikki mikrokontrollerien salaukset voidaan murtaa, mutta tärkeää on tehdä siitä taloudellisesti kannattamatonta potentiaaliselle hyökkääjälle [5]. Muistiin tai suorittimeen voidaan sähkömekaanisesti liittyä joko erillisen liittimen tai testipisteiden kautta. Testipisteiden käyttö on nopeaa eikä kuluta liitintä mekaanisesti, mutta ne tarvitsevat piirilevykohtaisen neulapetimekaniikan, joka ohjaa kortin testipisteet oikeisiin neuloihin. Muistipiiri voi olla myös irrotettava, kuten muistikortti tai muistitikku, jolloin se voidaan ohjelmoida erikseen ja kiinnittää ohjelmoituna ohjauskortille.

3 Flash-muisti

Uudelleenohjelmoitavat muistit voidaan jakaa haihtuviin ja haihtumattomiin muistityyppeihin. Esimerkiksi tietokoneen käyttömuistina käytetty *muuttuva satunnaisosoitusmuisti (DRAM, engl. Dynamic Random Access Memory)* on haihtuva. Tällaiset muistit unohtavat niihin varastoidun tiedon ajan kuluessa, joten muisti täytyy virkistää säännöllisesti. Nykyaikaista tietokoneen käyttömuistia joudutaan päivittämään 128 000–1 024 000 kertaa sekunnissa muistipiirin asetuksista ja ympäristön lämpötilasta riippuen [6, s. 35–36].

Haihtumattomat muistityypit, joihin Flash-muisti kuuluu, säilyttävät tiedon pitkään. Bez [7] pitää yhtenä haihtumattomuuden kriteerinä sitä, että tieto säilyy muistissa vähintään kymmenen vuotta. Hän huomauttaa kuitenkin, että muisti voi vikaantua aiemminkin. Voidaan todeta, että Flash-muisti voi säilyttää tiedon pitkään oikein käytettynä, mutta jos sen vikaantumismekanismeja ei tunneta ja niitä huomioida, siltä ei voida vaatia pitkää käyttöikää.

Tämän luvun näkökulma on elektroniikkasuunnittelija, joka käyttää Flash-muistipiirejä laajemmassa sovelluksessa. Mikäli lukija on kiinnostunut aiheesta syvällisemmin, Micheloni [8], Brewer [9] ja Campardo [10] käsittelevät Flash-teknologiaa huomattavan yksityiskohtaisesti puolijohteista kotelointiin asti.

Luku alkaa Flash-muistin rakenneosista ja toimintaperiaatteista. Kirjoitus- ja tyhjennysoperaatiot ovat monimutkaisempia, joten niille on varattu oma luku. Sen jälkeen tutustutaan nykyaikaisten Flash-muistipiirien ominaisuuksiin komponenttien valmistajien tarjoaman tiedon perusteella. Lopuksi peilataan valmistajien julkaisemia tietoja teoriaan, erityisesti kiinnittäen huomio luotettavuuteen ja rajapintoihin.

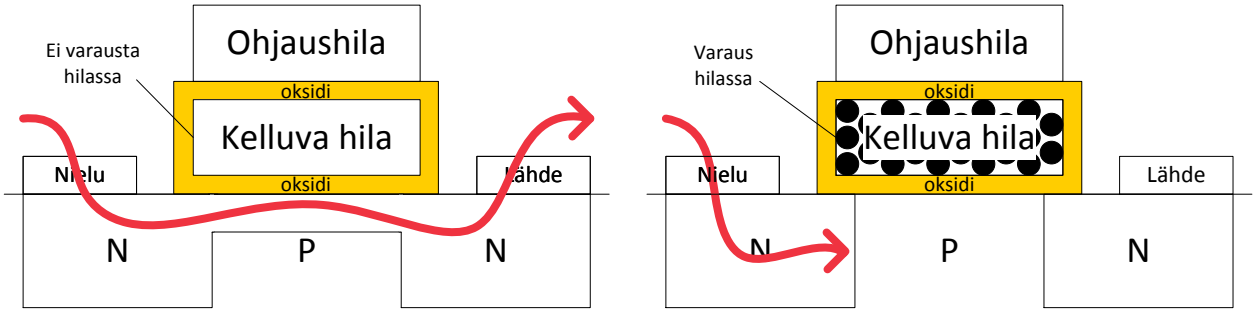
3.1 NOR- ja NAND-muistien perusteet

Flash-muistityypeistä kaksi on selkeästi muita yleisempiä: NAND ja NOR. Myös muita Flash-tyyppejä on olemassa ja uusia kehitetään edelleen, mutta ne eivät ole saaneet laajaa kaupallista suosiota. Niille on vaikeampi löytää vaihtoehtoisia piirejä kuin NAND- ja NOR-muisteille, joten niitä ei käsitellä tässä. Elektroniikkasuunnittelussa pyritään löytämään kaikille komponenteille niin samankaltainen vaihtoehto, että jos yhden vaihtoehdon suhteen ilmenisi saatavuusongelmia, se ei hidasta lopullisen sovelluksen tuotantoa. Aritome [11] kertoo Flash-muistin historiasta: NOR-tyyppinen Flash-muisti keksittiin Toshiballa vuonna 1984, ja sama tutkimusryhmä kehitti NANDin vuonna 1987 muistikapasiteetin lisäämiseksi. Flash-muistin rakenne muistuttaa huomattavasti *sähköisesti tyhjennettävää lukumuistia (EEPROM, engl. Electronically Erasable Programmable Read-Only Memory)* ja Flashiä voidaan kutsua myös nimellä Flash EEPROM. Periaatteellinen ero näiden välillä on, että EEPROM-muistit voidaan tyhjentää tavu kerrallaan, kun taas Flash-muistit tyhjennetään suuremmissa yksiköissä. *Flash (suom. välähdys)* on saanut nimensä siitä, että suuri muistialue voidaan tyhjentää nopeasti.

Tieto Flash-muisteissa on organisoitu erikokoisiin alueisiin, joissa jokainen hierarkiassa korkeammalla oleva alue sisältää useita matalammalla olevia. Tarkat nimitykset vaihtelevat hieman komponentti- tai valmistajakohtaisesti ja jotkin sisältävät lisäalueita. Yleisesti hierarkia noudattaa seuraavaa jakoa suurimmasta alueesta pienimpään: koko

muistipiiri, lohko[3], sivu, tavu tai sana ja solu. Solu on kooltaan yksi bitti, tavu on kahdeksan bittiä ja sana on kaksi tavua. Sanoja käytetään vain sähköiseltä rajapinnaltaan 16 rinnakkaisen signaalin, eli yhden sanan, muisteissa. Sivun koko on tavallisesti NOR-Flasheissä alle kilotavu ja NANDeillä muutama. Lohkon koko on usein noin kaksi dekadia sivua suurempi. Pienin tyhjennettävä muistialue on yksi lohko. NAND-muisteilla pienin kirjoitettava alue on yksi sivu, mutta jotkin muistit tarjoavat ominaisuuden kirjoittaa vain osan sivusta [8, s. 66–67]. NOReilla voidaan kirjoittaa yksittäisiä tavuja, mutta monet luvussa 3.3 käsiteltävistä muistipiireistä sisältävät sivun kokoisen puskurin, minkä vuoksi korkein ohjelmointinopeus saavutetaan kirjoittamalla sivu kerrallaan.

Pienin mahdollinen tiedon ala, solu, muistuttaa *kanavatransistoria (FET, engl. Field Effect Transistor)*. Kuvassa 3 on yleinen Flash-muistin solu. Suurin ero kanavatransistoriin on *kelluva hila (engl. Floating Gate)*, joka toimii muistisolun tietovarastona. Se on hyvin eristetty, joten sen tila ei ideaalisesti muutu ilman ulkoisia toimia. Kelluva hila on kanavatransistorista tutun ohjaushilan ja ohjattavan kanavan välissä joten se vaikuttaa kanavan ohjaukseen nostamalla vaadittavaa hilan kynnysjännnitettä[4]. Solun lukeminen toimii siten, että ohjaushilalle annetaan ennalta tunnetun kynnysjännitteen perusteella tietty jännite ja mitataan kanavan läpi kulkeva virta. Jos virta kulkee kanavan läpi, kelluvassa hilassa ei ole varausta estämässä hilan ohjausta. Mikäli kelluvassa hilassa on varaus, kanavassa ei kulje virtaa. Johtamisen tulkinta bittiarvoksi on käytäntökysymys, mutta yleisesti tyhjennetylle, johtavalle, solulle annetaan bittiarvo yksi, ja kirjoitetulle, varaukselliselle, solulle annetaan arvo nolla [9, s. 64].



Kuva 3: Flash-muistin muistisolussa tieto tallentuu kelluvaan hilaan.

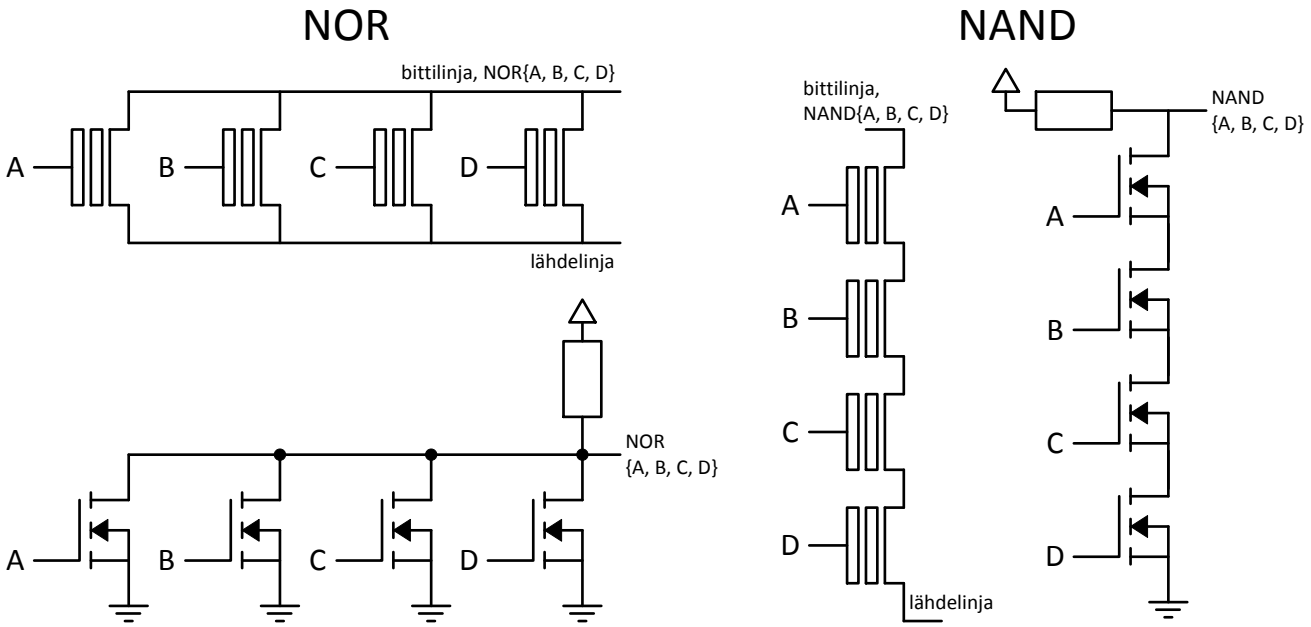
Edellä selitetty binäärimuotoinen *yksitasoinen solu (SLC, engl. Single-Level Cell)* voidaan laajentaa *monitasoiseksi soluksi (MLC, engl. Multi-Level Cell)* varastoimalla kelluvaan hilaan varausmäärä, joka on binääriarvojen yksi ja nolla välissä. Tällöin solu sisältää useita bittejä informaatiota. Monitasoisuus asettaa kuitenkin teknologisia haasteita kirjoittamisen ja lukemisen tarkkuudele sekä tiedon säilymiselle, joten toistaiseksi monitasoisella muistilla viitataan tavallisesti kaksitasoiseen muistiin [7].

[3]Lähteestä riippuen, lohkoa saatetaan kutsua myös sektoriksi, mikä on yleistä erityisesti NOReilla.

[4]*Kynnysjännite (engl. threshold voltage)* on jännite, joka määrää johtaako kanava vai ei. Kun hilan jännite on suurempi kuin kynnysjännite, kanavaan kerääntyy riittävästi elektroneja ja kanava johtaa. [12, s. 238–240]

Yksinkertaisuuden vuoksi, jos ei toisin mainita, tässä luvussa oletetaan muistien olevan yksitasoisia, eli että yhteen soluun tallennetaan vain yksi bitti tietoa.

NOR- ja NAND-nimet[5] tulevat samannimisistä Boolen logiikan porteista. Kuvassa 4 yhden bittilinjan muistisolut on järjestetty samalla tavalla kuin jos logiikkaportti tehtäisiin kanavatransistoreilla, joten kuvat ovat lähes identtiset. Toiminta on myös samanlaista: NORin tapauksessa bittilinjan (ulostulo) arvo määräytyy siten, että luettavan solun ohjaushila nostetaan ylös ja muiden lasketaan. Tällöin luettavan solun kelluvan hilan varaus ja sen asettama kynnysjännite määrää täysin ulostulon arvon. Myös NAND-muistin rakenne ja samanniminen logiikkaportti ovat lähes identtisiä. Tässä tapauksessa bittilinjan arvo määräytyy siten, että luettavan solun hilajännite nostetaan sellaiselle tasolle, jossa kelluvan hilan varaus määrää solun ja bittilinjan arvon, ja muiden solujen hilajännite nostetaan niin korkealle, että ne johtavat varmasti [13]. Käytännössä NAND-bittilinjalle kuuluu myös kaksi valintatransistoria, joilla se aktivoidaan. Näillä ei kuitenkaan ole merkitystä keskusteltaessa ainoastaan yhdestä linjasta, joka oletetaan aktiiviseksi, joten ne on jätetty kuvasta pois selkeyden vuoksi.



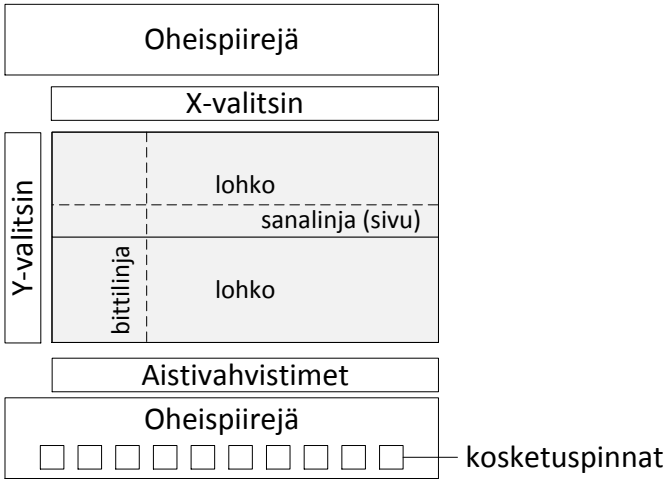
Kuva 4: NOR-tyypin rakenne vasemmalla ylhäällä muistuttaa kanavatransistoreista tehtyä NOR-porttia vasemmalla alhaalla. NAND-Flashin rakenne keskellä ja kanavatransistoreilla tehty looginen NAND-portti oikealla muistuttavat huomattavasti toisiaan.

Yksittäisten bittilinjojen suhde kokonaisuuteen selviää kuvasta 5. Kuva esittää Flash-muistin eri osien sijoittelua kotelon sisällä. NAND- ja NOR-muistien rakenne on tällä tasolla tarkasteltuna käytännössä identtinen ja erot ovat ”Oheispiirejä”-lohkossa sekä soluissa. Kuvassa bittilinjoja on pystysuunnassa vierekkäin. *X- ja Y-valitsimet*[6]

[5]NOR eli TAI-operaation negaatio, ”EI-TAI”. NAND eli JA-operaation negaatio, ”EI-JA”.

[6]Valitsimia nimitetään usein kirjallisuudessa sarake- ja rivivalitsimiksi, mutta muistivalmistajien datalehdissä niistä käytetään yleisesti nimitystä X- ja Y-valitsimet. Termit tarkoittavat samaa asiaa.

(engl. X and Y decoders) hallitsevat solujen, ja NANDissä lisäksi valintatransistorien, hilajännitteitä ja täten valitsevat luettavat bitit, joiden läpi kulkevan virran *aistivahvistin (engl. sense amplifier)* muuntaa digitaaliseksi signaaliksi. Lopuksi tulos viedään piirin kotelon pinneille *kosketuspintojen (engl. pad)* kautta.



Kuva 5: Flash-muistin korkean tason rakennekaavio, joka yhdistää NAND- ja NOR-tyyppien yhteiset piirteet [8, s. 22, 67][10, s. 151–158].

Kirjoittaminen ja tyhjentäminen ovat lukemiseen verrattuna monimutkaisempia operaatioita ja niitä käsitellään tarkemmin luvussa 3.2. Lyhyesti NOR- ja NAND-muistit käyttävät erilaisia kirjoitusmenetelmiä: NOR käyttää *kuumakanavaelektronimenetelmää (CHE, engl. Channel Hot Electron)* ja NAND Fowler–Nordheim-tunnelointia. Molemmat muistityypit käyttävät Fowler–Nordheim-tunnelointia tyhjennysoperaatiossa. Muistin tilaa muokkaavat operaatiot perustuvat voimakkaiden sähkökenttien käyttöön.

NOR-Flashin erityisominaisuus on nopea *satunnaisen paikan lukeminen (engl. Random Read Access)*, minkä ansiosta sovellus voidaan ajaa suoraan Flash-muistista ilman että se ladataan erilliseen käyttömuistiin. Tätä ominaisuutta kutsutaan *suora-ajoksi (XIP, engl. eXecute-In-Place)*. [14] NORin luotettavuus on myös NANDiä parempi, jonka vuoksi NORin kanssa ei tarvitse käyttää virheenkorjausta, mutta asiaan palataan tarkemmin luvussa 3.4.

NAND kehitettiin nimenomaan edulliseksi massamuistiksi, minkä vuoksi muistisolut on järjestelty niin, että ne käyttävät vähemmän pinta-alaa kuin NOR-muistit. Tarkat lukuarvot eri Flash-tyyppien pinta-alojen suhteesta vaihtelevat eri lähteissä, mutta yksi useassa lähteessä esitetty luku on, että NAND-solu on noin 40 % NOR-solun pinta-alasta [9, s. 16, 71][14][15]. Koska solut ovat bittilinjassa jonomaisesti, virran – eli bitin – lukeminen luotettavasti usean muistisolun läpi on hidasta, joten NAND-muisteilla tietoa käsitellään suuremmissa yksiköissä laskennallisen bittikohtaisen käsittelynopeuden kasvattamiseksi [16]. NANDien satunnaisosoitusnopeus on jopa kolme dekadia hitaampi kuin NOReilla, mutta tarvittaessa ohjelmakoodia voidaan suorittaa NANDiltä käyttämällä haihtuvaa välimuistia, jolloin voidaan saavuttaa jopa NORin suora-ajoa parempi sovelluksen suoritusnopeus [17]. Van Houdt [14] mainitsee NANDin kirjoitukseen käytettävän Fowler–Nordheim tunneloinnin vaatimien piirien

vievän vähemmän tilaa kuin mitä NORin CHE-menetelmä tarvitsee. Tilaa säästetään luotettavuuden kustannuksella, minkä vuoksi NAND-muisteilla joudutaan käyttämään erillistä virheenkorjausta riittävän luotettavuuden saavuttamiseksi. Tähän palataan myöhemmin luvussa 3.4.

Nykyään Flash-muisteja käytetään muista digitaalipiireistä tutuilla *yksipuolisilla (engl. single supply)* käyttöjännitteillä. Yleisimpiä NAND- ja NOR-malleja on saatavilla alkaen 1,65 V jännitteistä aina 5 V asti. Muistien kirjoitus- ja tyhjennysoperaatiot vaativat kuitenkin huomattavasti näitä suurempia jännitteitä, jopa 10–20 V. Tarvittavat jännitteet luodaan varauspumpputekniikoilla[7]. Ensimmäisissä Flash-muisteissa oli 5 V käyttöjännitteen lisäksi erillinen 12 V jännite muistin tilaa muuttaville operaatioille, jolloin varauspumppuja ei tarvittu [7][16]. Joissain muistipiireissä on edelleen pinni korkeampaa jännitettä varten kirjoitus- ja tyhjennysoperaatioiden nopeuttamiseksi, mutta sen käyttö ei ole pakollista.

Flash-piirien kapasiteetti pinta-alaa kohden on kehittynyt nopeammin kuin tuotantotekniikka [18]. Luonnollisesti valmistusprosessin muuttuessa tarkemmaksi, muistisoluja voidaan valmistaa enemmän samalle pinta-alalle. Monitasoisten solujen ja piirin sisäisen rakenteen optimointien avulla muistipiirin kapasiteettia on voitu kasvattaa edelleen. Fyysisesti pienempiin soluihin mentäessä, molemmilla muistityypeillä ongelmaksi muodostuu viereisten solujen aiheuttama häiriö, joka muuttaa kelluvan hilan kynnysjännitettä. Lisäksi oksidikerroksen pienentyessä solun varaus muuttuu helpommin tahattomasti, vaikka juuri kaiken ollessa pienempää, myös elektroneja – ja siten muutosvaraa – on vähemmän. Viime vuosina oksidikerrosta ei olekaan enää voitu pienentää tästä syystä. [14] Flash-alalla pidetään todennäköisenä, että nykyisenlaista muistisolua ei voida pienentää enää kovin pitkään, mutta sen sijaan kapasiteetin odotetaan kasvavan edelleen. Mikolajick [19] esittelee lupaavimpia teknologioita, joilla Flash-piirien kehitystä voidaan jatkaa. Näitä ovat esimerkiksi entistä parempi oksidimateriaali, erilainen solurakenne sekä kolmiulotteinen muistipiiri. Näistä viimeisimmällä tarkoitetaan muistisolujen valmistamista kerroksittain päällekkäin, kun nykyään Flash-piirit valmistetaan käytännössä kaksiulotteisesti. On huomattava, että nykyäänkin on olemassa Flash-muisteja, joissa kokonaisia Flash-piirejä on pinottu päällekkäin kapasiteetin kasvattamiseksi, mutta kolmiulotteinen Flash-piiri tarkoittaa tarkemmin piiriä, jossa vain muistisolut ovat päällekkäin, mutta tarvittavat oheispiirit jaetaan ja täten säästetään pinta-alaa.

Yhteenvetona todetaan, että kumpikin yleisimmistä muistityypeistä on selkeästi suunniteltu erilaisiin käyttötarkoituksiin. NOR-Flashit soveltuvat hyvin sovelluksiin, jossa satunnaisessa sijainnissa olevaan pieneen määrään tietoa täytyy päästä käsiksi nopeasti. Toisin sanoen se soveltuu ohjelmiston tallentamiseen ja suorittamiseen. Tiheämmät NAND-Flashit puolestaan on suunniteltu massamuisteiksi. Ominaisuuden kääntöpuoli on, että tallennettua tietoa käsitellään suurehkoissa osioissa. Yhteistä molemmille muisteille on, että ne ovat haihtumattomia muisteja, jotka tyhjennetään suurissa lohkoissa.

[7]Syvällisempi katsaus piiritason ratkaisuista on saatavilla Breweriltä [9, s. 114–118] ja NANDien algoritmiratkaisuista Michelonilta [8, s. 329–337].

3.2 Kirjoitus- ja tyhjennysoperaatio

Edellisessä luvussa käsiteltiin muistien lukuoperaatiota, mutta vain pinnallisesti kirjoitus- ja tyhjennysoperaatioita. Tässä luvussa tutustutaan muistin tilaa muokkaaviin operaatioihin yksityiskohtaisemmin ja tarkastellaan tekijöitä, jotka vaikuttavat näiden operaatioiden nopeuteen.

Jäljemmän tekstin kannalta on olennaista ymmärtää, että kelluvaan hilaan varastoituneiden elektronien muodostama varaus voi saada lukuisia arvoja. Campardo [10, s. 11] havainnollistaa tätä seuraavalla laskutoimituksella: oletetaan kelluvalle hilalle tyypillinen kapasitanssi $C_T = 0{,}7\,\mathrm{fF}$ ja oletetaan kelluvan hilan varastoimien elektronien aiheuttavan kynnysjännitteeseen $\Delta V_T = 3\,\mathrm{V}$ muutoksen, kun solu on kirjoitetussa tilassa. Tällöin kelluvan hilan varaus on

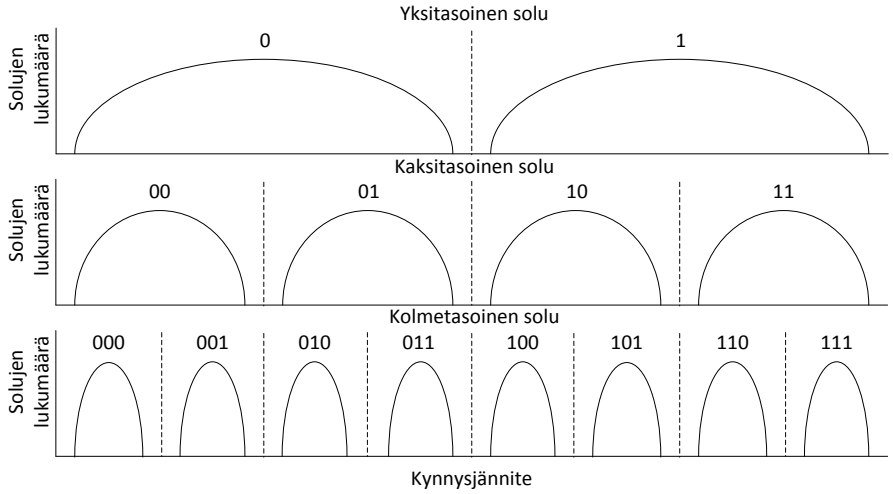
$$Q = C_T * \Delta V_T = 0{,}7\,\mathrm{fF} * 3\,\mathrm{V} = 2{,}1\,\mathrm{fC}, \tag{1}$$

joka elektronien lukumääränä on

$$\frac{2{,}1\,\mathrm{fC}}{1\,\mathrm{e}} = \frac{2{,}1 \cdot 10^{-15}\,\mathrm{C}}{1{,}6 \cdot 10^{-19}\,\mathrm{C}} \approx 13\,100\,\mathrm{elektronia}.^{8} \tag{2}$$

$1\,\mathrm{e}$ on alkeisvaraus. Esimerkistä ymmärretään heti, että kelluvalla hilalla ei ole yksittäistä ”tyhjää” ja ”täyttä” tilaa, vaan voidaan antaa ainoastaan rajat, joiden väliin sijoittuneesta varauksesta pystytään päättelemään kelluvan hilan tila. Vaikka varaus on selkeästi kvantittunut, elektronien suuren lukumäärän vuoksi virhe ei ole suuri, jos mieltää tämän jakauman jatkuvana. Yleisesti tässä tekstissä oletetulla yksitasoisella solulla on kaksi tilaa, yhdellä bitillä ilmaistuna 0 ja 1. Tällaisen solun varaus muodostaa siis kaksi jakaumaa, kuvan 6 mukaisesti. Varauksen hajanaisuus tai levinneisyys johtuu muun muassa valmistusprosessin epäideaalisuuksista, solujen välisestä kapasitanssista [8, s. 78–80] ja muistipiirissä käytettyjen aistivahvistinten tarkkuudesta [10, s. 359]. Usein kirjallisuudessa jakaumat on esitetty normaalijakaumina tai jakaumina, joiden huippu on keskellä. Käytännössä muoto ei välttämättä ole näin symmetrinen, vaan oikeastaan ramppimuotoinen, mihin vaikuttaa myös käytetty kirjoitus- tai tyhjennysmenetelmä [7][20]. Tässä luvussa käsiteltävät asiat voidaan suoraan yleistää n-tasoiselle, eli n-bittiselle, muistisolulle jakamalla sallittu varausalue 2^n osaan. Toisin sanoen kaksitasoisella muistilla on neljä ja kolmitasoisella kahdeksan jakaumaa, joten monitasoisilla muisteilla jakaumien tulee olla huomattavasti yksitasoista kapeampia, kuten kuvasta 6 nähdään. Mitä kapeampiin jakaumiin mennään, sitä hankalammaksi, ja täten hitaammiksi, muistin tilaa muokkaavat operaatiot muuttuvat, ja entistä vähemmän elektroneja saa karata tahattomasti kelluvasta hilasta. Vuonna 2005 ennustettiin, että 2010-luvun aikana tullaan siirtymään 30 nm valmistusprosessiin, jolloin solun kapasitanssin pienetessä kelluvaan hilaan varastoidaan enää satoja elektroneja, joten jo yksittäisten elektronien karkailut ovat merkittävä luotettavuusongelma monitasoisilla soluilla [18]. Luvussa 3.3 esitetyt nykyisin myytävät muistit on valmistettu 65–110 nm prosesseilla.

[8] Campardo [10, s. 11] laski ”$2{,}1\,\mathrm{fC} \approx 2{,}1 \cdot 10^{-15} \cdot 1{,}6 \cdot 10^{19} = 33\,600\,\mathrm{elektronia}$”, mikä on mielestäni laskuvirhe, koska $\frac{1}{x \cdot y^{-z}} \neq x \cdot y^z$, eli termi $1{,}6$ kuuluu nimittäjään. Kuitenkin merkittävää on, että elektronien lukumäärä lasketaan tuhansissa.



Kuva 6: Solujen kynnysjännitteen vaihtelu, kun muistiin on kirjoitettu jokaista bittiarvoa yhtä monta kappaletta. Piikkien sijaan kynnysjännite jakautuu leveämmälle alueelle. [8, s. 455–459] Todellisuudessa jakaumat eivät välttämättä ole yhtä symmetrisiä.

NOR-muisteilla varaus syötetään hyvin eristettyyn kelluvaan hilaan *kuumakanavaelektronimenetelmällä (CHE, engl. Channel Hot Electron)* ja NAND-muisteilla *Fowler–Nordheim-tunneloinnilla (FN)*. CHE toimii siten, että kanavan nielulle asetetaan keskivoimakas (≈ 4 V) ja ohjaushilalle voimakas jännite (≈ 10 V) samalla pitäen lähde maapotentiaalissa. Nyt kanavassa kulkee jopa 1 mA virta, joka antaa osalle elektroneista niin suuren energian, että ne voivat injektoitua kelluvaan hilaan. CHE-injektio ei ole suuren virran ja jokseenkin suuren jännitteen vuoksi tehonkulutuksen kannalta kovin hyvä operaatio, mutta se on erittäin nopea; yksittäisen solun kirjoitusaika kestää noin mikrosekunnin. [20][9, s. 64, 183] FN-tunneloinnilla kirjoitettaessa ohjelmoitavan solun ohjaushilalle asetetaan korkea jännite (≈ 20 V) kanavan ollessa maapotentiaalissa. Muille saman bittilinjan ohjaushiloille asetetaan niin suuri jännite (≈ 10 V), että ne johtavat, mutta eivät aiheuta vielä FN-tunnelointia. Riittävän suuren sähkökentän muodostuessa, elektronit pystyvät tunneloitumaan kelluvaa hilaa ympäröivän oksidikerroksen läpi. FN-tunneloinnilla elektronit läpäisevät oksidin CHE-injektiota huonommin, joten NANDeillä joudutaan käyttämään NORia ohuempaa oksidikerrosta ja korkeampia jännitteitä, eli suurempia sähkökenttiä. Molemmat näistä tekijöistä edistävät kelluvan hilan varauksen spontaania muuttumista. [9, s. 228–229] Varauksen itsestään muuttumiseen ja sen vaikutuksiin palataan kappaleessa 3.4. FN-tunnelointi on CHE-injektiota hitaampi operaatio, mutta virraksi riittää jopa alle 1 nA, joten useita soluja voidaan kirjoittaa rinnakkain.

Kirjoittamisen vastakohta on muistisolun tyhjentäminen, mikä sekä NAND- että NOR-muisteilla tapahtuu FN-tunneloinnilla. Vaikka perusperiaate on sama, yksityiskohdat käytetyissä jännitteissä sekä kohdissa, joiden potentiaalia nostetaan tai lasketaan, eroavat hieman. NAND-tyypillä ohjaushilalle asetetaan 0 V jännite ja kanavan P-tyypin substraatin, katso kuva 3, jännite nostetaan 20 V asti [8, s. 227–228]. NOReilla taas nielun jännitteeksi asetetaan ≈ 5 V ja ohjaushilan jännitteeksi -10 V, jolloin vältytään suurilta yksittäisiltä jännitteiltä [9, s. 64]. Hieman erilaisilla toimenpiteillä saavutetaan sama FN-tunnelointi, jolla tyhjennetään kokonaisia lohkoja kerrallaan.

Muistisolun epäideaalisuuksista, kuten oksidin pilaantumisesta (katso luku 3.4), johtuen samanlainen kirjoitus- tai tyhjennysoperaatio ei tuota samaa lopputulosta koko muistin elinikää, joten muistin tilan muuttamiseen käytetään monimutkaisempia algoritmeja. Molempien operaatioiden käyttämät algoritmit noudattavat samaa perusideaa, joten käsitellään tässä esimerkkinä ainoastaan kirjoitusalgoritmia. Hyvin yksinkertaistettuna algoritmi on

Kunnes solun varaus $\geq$ tavoitearvo
    ohjelmoi hetken aikaa
    tarkista solun varaus
    nosta ohjelmointijännitettä
Toista

Yksityiskohtaisemmin ensimmäinen ohjelmointikierros on tavallisesti muita pidempi ja jo pelkästään sillä on tarkoitus saavuttaa haluttu arvo. Ohjelmointijännite ei ole vakio, vaan sitä muutetaan hilan varauksen mukaan siten, että aiheutuvan sähkökentän voimakkuus on mahdollisimman vakio.[9] Koska muistisolun kynnysjännite muuttuu lähes lineaarisesti, myös ohjelmointijännite muuttuu samassa suhteessa, käytännössä porrasmaisesti approksimoituna. [8, s. 62–64]. Algoritmin iteratiivisen luonteen vuoksi mikä tahansa varausmäärä on mahdollista saavuttaa helposti, mutta monitasoisilla soluilla tilan muuttaminen on hitaampaa, koska pienempien virhemarginaalien vuoksi joudutaan käyttämään lyhyempiä ohjelmointiaskelia vaaditun tarkkuuden saavuttamiseksi.

Ohjelmointialgoritmin kokonaissuoritusajan tyypillinen ja maksimikesto on ilmoitettu datalehdissä, mutta operaation pilkkomisesta yksityiskohtaisemmiksi osiksi on kirjoitettu vain vähän. Vuonna 1995 Suh [13] vertaili edellä esitettyä ohjelmointialgoritmia sen aikaiseen tavallisesti käytettyyn kirjoitusalgoritmiin NAND-muisteilla. Hän jakoi yhden 40 μs kestävän ohjelmointikierroksen neljään osaan: sopivien jännitteiden asettaminen bittilinjalle (8 μs), varsinainen kirjoitus (20 μs), jännitteiden purkaminen (4 μs) ja ohjelmointituloksen tarkistus (8 μs). Näitä kierroksia tarvittiin tavallisesti 4–7 oikean kirjoitustuloksen saavuttamiseksi, mikä tarkoittaa, että ohjelmointi kestää noin 200 μs. Nykyään myytävillä NAND-muisteilla ohjelmointiajaksi mainitaan tavallisesti 200–700 μs, joten Suhin mainitsemat lukuarvot ovat nykyäänkin suuntaa-antavia. Nykymuistien korkeampi ohjelmoinnin maksimikesto voidaan selittää luvussa 3.4 käsiteltävällä oksidin pilaantumisella, jonka vuoksi useampia ohjelmointikierroksia joudutaan suorittamaan oikean arvon saavuttamiseksi. Koska yksitasoisilla muisteilla solun varaus saa asettua laajemmalle alueelle, ensimmäisellä ohjelmointikierroksella voidaan kirjoittaa jäljempiä kierroksia pidempään ja saavuttaa tavoitearvo nopeammin kuin monitasoisilla muisteilla [10, s. 349–351]. Sivukoon pysyessä samana, kaksitasoisella NAND-muistilla ohjelmointiaika voi olla nelinkertainen yksitasoiseen NANDiin verrattuna [21].

Edellisestä selityksestä jätettiin operaatioiden *rajoittaminen (engl. inhibit)* mainitsematta yksinkertaisuuden vuoksi. Kyseessä on oleellinen toimenpide muistin oikean toiminnan kannalta. NAND-muisteilla muistisolun arvoa luettaessa muiden samassa bittilinjassa olevien solujen ohjaushilan jännite nostetaan niin korkealle, että ne varmasti johtavat. Kuitenkin jos kelluvassa hilassa on liikaa varausta, on mahdollista, etteivät muut solut

[9] Ennen 2000-lukua ohjelmointijännite pidettiin tavallisesti vakiona, jolloin ohjelmointikierroksia tarvittiin jopa 5–8 kertaa niin monta kuin jos ohjelmointijännitettä nostetaan kierros kierrokselta [13].

(yksi tai useampi) johda ja lukuoperaatio epäonnistuu. Tätä kutsutaan *liikakirjoittamiseksi (engl. over-programming)*. NORien rakenteesta johtuen niillä ongelma on käänteinen: jos solua tyhjennetään liikaa ja solun kynnysjännite menee negatiiviseksi, solu saattaa johtaa ilman ohjausjännitettä, mikä aiheuttaa virheellisen lukemisen. Tätä kutsutaan *liikatyhjennykseksi (engl. over-erase)*. Solujen epäideaalisuuksien vuoksi ne vastaanottavat tai luovuttavat varauksen eri nopeuksilla. Siksi on tärkeää, että kun solun varaus on toivotulla alueella, sen ohjelmointi tai tyhjennys lopetetaan. Operaatioita ei pidä aloittaa alun perinkään niille soluille, joilla on jo toivottu arvo: esimerkiksi kirjoitettaessa tavu $1010\,1010$ tyhjään muistiin ($1111\,1111$), joka toisen bitin arvo on jo tavoitearvo (1), joten niitä ei muuteta. Micheloni [8, s. 67–76] luettelee NAND-muisteille erilaisia keinoja kirjoituksen rajoittamiseksi, mutta kaikkien pääperiaate on kuitenkin sama: lasketaan kelluvan hilan yli olevaa jännitettä. Eräs yksinkertainen menetelmä on *itsetehostettu kirjoituksen esto (SBPI, engl. Self-Boosted Program Inhibit)*. NAND-solua kirjoitettaessa ohjaushilalla on ≈ 20 V ja kanavan jännite on maapotentiaalissa, siten kelluvan hilan yli on 20 V jännite. SBPI:tä käytettäessä kanavan jännitteeksi asetetaan esimerkiksi 8 V, jolloin kelluvan hilan yli on enää 12 V jännite, joka ei riitä FN-tunnelointiin ja ohjelmointia ei tapahdu. NOR-muisteilla ja tyhjennettäessä periaate on sama. Muistisolujen liikakirjoitus ja -tyhjennys sekä väärien muistisolujen kirjoitus estetään nostamalla jännitettä sopivista kohdista niin, ettei muistisolun yli muodostu merkittävää jännitettä.

Tässä luvussa tutustuttiin NAND- ja NOR-muistien kirjoitus- ja tyhjennystoimintoihin. Operaatiot ovat muistin lukemista monimutkaisempia. Lisäksi ne ovat iteratiivisia, eli lyhyttä kirjoitus- tai tyhjennyskierrosta toistetaan kunnes saavutetaan haluttu lopputulos. Muistisoluun tallennettu varaus ei ole tarkka arvo, vaan se sijoittuu sille määritettyjen rajojen sisään.

3.3 Katsaus nykyaikaisiin Flash-muisteihin

Teorian perusteella on mahdollista selittää Flash-muistien toimintaperiaate ja olennaiset erot NAND- ja NOR-tyyppien välillä. Teoreettisen tarkastelun ongelma on, ettei se anna ajantasaista kuvaa muistien todellisesta suorituskyvystä ja ominaisuuksista nopeasti kehittyvällä alalla, koska osassa julkaisuista käsitellään jo vanhentunutta tietoa ja osassa niin uutta, ettei kaupallisia sovelluksia vielä ole. Tässä luvussa tutustutaan työn kirjoitushetkellä markkinoilla oleviin Flash-muistipiireihin niiden datalehtien perusteella. On huomattava, että seuraavassa esitetty tieto ei sisällä koko Flash-markkinaa kattavasti, vaan sen perusteella voidaan ainoastaan esittää suurehkoja linjoja ja johtopäätöksiä komponenteista, joita myydään tällä hetkellä.

Tarkasteltavat muistit valittiin kolmella perusteella. Ensimmäiseksi tarkasteluun otettiin pääasiassa suurimpien valmistajien muisteja. Valmistajien kokoina käytettiin vuoden 2012 markkinaosuuksia, jotka on tiivistetty taulukkoon 1. Myös muutama muu valmistaja on mukana monipuolisuuden vuoksi. NAND-muisteilla tämä on erityisen oleellista, koska Samsung Electronicsin, SK Hynixin ja Intelin Flash-muistien datalehdet sekä osa Micronin ja Toshiban vastaavista eivät olleet vapaasti saatavilla[10]. Toinen peruste on muistipiirin tallennuskapasiteetti. Työn toimeksiantajalle riittää toistaiseksi

[10]Jotkin ei-vapaista datalehdistä ovat saatavilla Internetissä, mutta vaativat salassapitosopimuksen hyväksymistä. Siksi niitä ei käsitellä tässä vapaasti julkaistavassa opinnäytetyössä.

64 Mb, joten vertailuun valittiin muistit joiden kapasiteetti on mahdollisimman lähelle 64 Mb, mutta ei vähempää. Kovin pieniä NAND-muisteja ei ole saatavilla, joten NANDeistä valittiin pienin tuoteperheeseen kuuluva. Kolmas kriteeri on, että koska yksi valmistaja valmistaa useita erilaisia tuoteperheitä, näistä pyrittiin valitsemaan erilaisia monipuolisuuden vuoksi.

Taulukko 1: NOR-muistien valmistajien markkinaosuudet vuoden 2012 toisella *vuosineljänneksellä (Q, engl. quarter of a year)* [22] ja NAND-muistien valmistajien markkinaosuudet koko vuonna 2012 [23]

NOR-valmistajat, Q2/2012

	Osuus [%]
Spansion	27,8
Micron	25,9
Macronix	14,3
Winbond	11,4
Eon Silicon	2,9
Muut	17,7
Yhteensä	100,0

NAND-valmistajat, 2012

	Osuus [%]
Samsung Electronics	36,9
Toshiba	30,8
Micron Technology	13,6
SK Hynix	11,4
Intel	7,1
Muut	0,3
Yhteensä	100,0[11]

Taulukkoon 2 on koottu 16:sta eri Flash-tuoteperheestä joukko avainparametrejä. Taulukko on jaettu pystyviivalla kolmeen osaan: sarjarajapintaiset NOR-muistit[12], rinnakkaisesti tietoa siirtävät vastaavat ja NAND-muistit. NORin kohdalla raja on häilyvä, koska kaikki vertailuun valitut sarjamuotoiset NOR-muistit tukivat rinnakkaisuutta ainakin jollain tasolla. Erityisesti Adesto Technologiesin muistin tapauksessa käyttäjä valitsee, haluaako hän käyttää sarja- vai rinnakkaismuotoista rajapintaa. Lisäksi taulukossa ominaisuudet on listattu samassa järjestyksessä kuin niitä käsitellään tekstissä.

Kaikki vertailtavat muistit jakavat joitain ominaisuuksia. Kaikkien käyttöjännite on 3 V (2,7–3,6 V). Lisäksi voidaan olettaa muistisolujen olevan yksitasoisia, eli että yhteen soluun tallennetaan vain yksi bitti tietoa. Yksitasoisuutta ei mainittu kuin muutamassa datalehdessä, mutta vertailun ulkopuolelle jääneistä datalehdistä huomattiin, että monitasoisilla soluilla sivun absoluuttinen sivun kirjoitusaika – ei siis aikayksikköä kohden – on moninkertainen yksisoluisiin verrattuna, kuten teoria antaa olettaa [9, s. 283–286].

[11] NAND-valmistajien markkinaosuuksien summa (100,0 %) perustuu pyöristysvirheettömiin lukuihin, sillä jos esitetyt prosentit lasketaan yhteen, summa on 100,1 %.

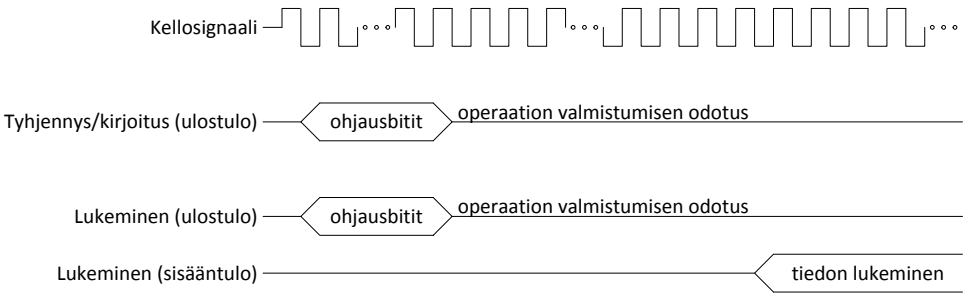
[12] Tarkempi sanamuoto on ”NORin kaltaiset muistit”, koska datalehti tai valmistajan Internet-sivusto ei aina suoraan mainitse muistin tyyppiä täsmällisesti, toisin kuin NANDin tapauksessa. Ominaisuuksiensa puolesta ne kuitenkin ovat niin NORin kaltaisia, että kaikkia käsitellään tekstissä NOReina.

Taulukko 2: Katsaus tällä hetkellä markkinoilla oleviin Flash-muisteihin

	1. Eon Silicon Solution EN25QH64	2. Macronix International MX25L6435E	3. Micron M25PX64	4. Micron N25Q064A	5. Spansion S25FL064P	6. Winbond W25Q64FV	7. Adesto Technologies AT45DB642D	8. Eon Silicon Solution EN29GL064	9. Macronix International MX29LV640E	10. Micron PC28F064M29EWXX	11. Spansion S29GL064N	12. Winbond W29GL064C	13. Eon Silicon Solution EN27LN1G08	14. Micron NAND128-A	15. Spansion S34ML01G1	16. Toshiba TC58DVG02D5TA00
Viite	[24]	[25]	[26]	[27]	[28]	[29]	[30]	[31]	[32]	[33]	[34]	[35]	[36]	[37]	[38]	[39]
Julkaisuvuosi	2011	2011	2012	2010	2008	2011	2005	2009	2008	2012	2007	2010	2011	2003	2012	2011
Datalehden revisio	H	1.1	B	J	08	F	M	N	1.7	B	12	F	B	17	08	1.10
Flash-tyyppi	-	NOR	NOR	NOR	NOR	-	DF[1]	-	NOR	NOR	NOR	-	NAND	NAND	NAND	NAND
Rajapintatyyppi[2]	S	S	S	S	S	S	S/R	R	R	R	R	R	R	R	R	R
Kapasiteetti [Mb]	64	64	64	64	64	64	64	64	64	64	64	64	1024	128	1024	1024
Satunnaisosoitus [ns]	880	390	320	330	250	270	500	70	70	60	90	70	$25{\cdot}10^3$	$12{\cdot}10^3$	$25{\cdot}10^3$	$25{\cdot}10^3$
Lukunopeus [Mt/s]	25,0	43,0	18,7	54,0	40,0	52,0	50,0	36,0	14,3	36,8	34,4	36,0	26,7	13,6	26,8	26,7
Kirjoitusnop. [Mt/s]	0,2	0,2	0,3	0,6	0,2	0,4	0,4	0,2	0,2	3,2	0,1	0,3	10,0	2,6	10,2	6,8
Tyhjennysnop. [Mt/s]	0,3	0,2	0,1	0,2	0,1	0,4	0,1	0,5	0,2	0,1	0,1	0,4	85,3	8,0	36,6	51,2
Rinnakkaisuus	x4	x4	x2	x4	x4	x4	x8	x16	x16	x16	x16	x16	x8	x8	x8	x8
Kirjoituksen kiihdytys	Ei	Ei	Kyllä	Kyllä	Kyllä	Ei	Ei	Kyllä	Kyllä	Kyllä	Kyllä	Kyllä	Ei	Ei	Ei	Ei
Tiedon säilyvyys [a]	20	20	20	20	20	20	20	20	20	20	20	20	10	10	10	-[3]
Uudelleenkirjoituksia	10^5	10^5	10^5	10^5	10^5	10^5	10^5	10^5	10^5	10^5	10^5	10^5	10^5	10^5	10^5	-[3]
Käskykanta	SFDP	SFDP	-	SFDP	CFI	SFDP	-	CFI	CFI	CFI	CFI	CFI	-	-	ONFI	-
Kirjoituksen tilatieto[4]	R	R	R	R	R	R	P&R	P	P	P&R	P	P&R	P&R	P&R	P&R	P
Hinta[5] [USD]	-	1,49	1,79	1,01	0,91	0,65	3,14	-	1,55	1,57	1,33	0,79	-	1,94	1,50	2,43
Hankintamäärä[5]	-	1 960	1 000	50	55	5	2 000	-	150	10 000	86	5 000	-	1	34	5 000
Muuta															1 b ECC	

[1]DataFlash on Atmelin kehittämä yksityinen muistityyppi, joka perustuu NOR-teknologiaan. Adesto Technologies omistaa nykyään Atmelin Flash-piirien oikeudet.
[2]Rajapintatyyppi: S = Sarja, R = Rinnakkainen
[3]Toshiban datalehti sisältää kuvaajan tiedon säilyvyydestä tyhjennyskertojen funktiona, mutta kuvaajassa ei ole lukuarvoja.
[4]Kirjoituksen tilatieto: P = Pinni, R = Rekisteri
[5]Hinnat on tarkistettu 6.8.2013 Octopart-hakukoneella. Hinnat ovat voimassa niiden alla olevalla komponenttien pienimmällä hankintamäärällä.

Luku-, kirjoitus- ja tyhjennysnopeus aikayksikköä kohden ei ole yksiselitteinen käsite Flash-muisteilla komponenttitasolla, minkä vuoksi taulukossa 2 esitetyt nopeudet ovat laskettuja maksimiarvoja. Itse asiassa vain harvassa datalehdessä on edes mainittu suorituskykyä tietomääränä aikayksikköä kohden, vaan tavallisesti mainitaan ainoastaan lukemisen maksimitaajuus, kirjoituksen kesto sivua kohti ja lohkon sekä komponentin tyhjennysaika. Nämä arvot eivät ole vertailukelpoisia eri komponenttien kesken, koska sivujen ja lohkojen koot vaihtelevat, ja lukeminen voidaan suorittaa eri tavoin. Erityisesti sarjamuotoisilla NOR-muisteilla lukemiseen voidaan käyttää perinteisen yksibittisen sarjakommunikaation lisäksi myös useampia pinnejä yhtäaikaisesti, jolloin tietoliikenne on käytännössä enemmän rinnakkaista. Rinnakkaisuudella saavutetaan huomattavasti suurempi tiedonsiirtonopeus, mutta se vaatii monimutkaisempaa ohjelmistoajuria. Lisäksi kaikentyyppiset muistit on mahdollista tyhjentää vähintään lohko- sekä komponenttitasolla, joilla saavutetaan erilaiset suorituskyvyt. Eräs suorituskykyä monimutkaistava asia on myös kirjoituksen ja tyhjennyksen kiihdytysjännite. Flash-komponentit tarvitsevat muistin tilaa muokkaaviin operaatioihin huomattavasti käyttöjännitettä suuremman jännitteen. Tavallisesti tämä luodaan piirin sisällä käyttäjän kannalta automaattisesti, mutta monet muisteista tarjoavat erillisen pinnin, johon voidaan kytkeä valmiiksi suurempi jännite, jolloin muistin tilaa muokkaavat operaatiot voidaan suorittaa nopeammin. Korkeamman jännitteen syötteenä ottava pinni on merkitty taulukkoon termillä ”Kirjoituksen kiihdytys”. Datalehtiin kirjatut kirjoituksen ja tyhjennyksen tyypilliset ja maksimiarvot voivat erota paljonkin: maksimiarvo voi olla jopa yli 30 kertaa niin suuri kuin tyypilliseksi kirjattu arvo [32]. Kaikki nämä muuttujat huomioiden, taulukkoon 2 on laskettu muistien maksimisuorituskyky megatavuina sekunnissa tyypillisillä arvoilla sekä mahdollisella kiihdytyksellä, minkä vuoksi todellinen suorituskyky voi olla laskennallista arvoa huonompi.



Kuva 7: Tavallisten muistioperaatioiden signaalimuodot isäntäjärjestelmän näkökulmasta. Ohjausbittien ja operaatioiden odotuksen vaatimat ajat eivät ole piirretty mittakaavassa. Tietyt lukuoperaatiot eivät välttämättä sisällä odotusta lainkaan, kun pisimmät tyhjennyskomennot kestävät yli minuutin.

Muistien nopeuksia kuvaavat arvot on laskettu seuraavassa luetelluilla kaavoilla. Yksinkertaisin ja kaikilla taulukon muistityypeillä samankaltaisin operaatio on muistin tyhjentäminen, mikä noudattaa yleistetysti kuvan 7 esittämää signaalimuotoa. Joillain muisteilla esitetyn signaalin lisäksi tulee ohjata myös tiettyjä pinnejä sekä tarkistaa tyhjennyksen tila joko erillisestä tilapinnistä tai muistipiirin rekisteristä. Nämä toimet eivät vaikuta oleellisesti suorituskykyyn. Kuvan ohjausbitit sisältävät tyhjennyskäskyn, esimerkiksi lohko- tai komponenttitason tyhjennys, mahdollisen lohkon osoitteen ja joillain kompo-

nenteilla tyhjennyksen aloituskäskyn. Yleisesti komento on pituudeltaan yhdestä kuuteen tavua ja ajallisesti ohjausbittien välittäminen kestää kymmenistä nanosekunneista yhteen mikrosekuntiin. Varsinainen lohkotason tyhjennysoperaatio lasketaan NOR-muisteilla sekunneissa ja NAND-muisteilla millisekunneissa. Suorituskyvyn kannalta ohjausbiteillä ei siis ole merkittävää vaikutusta tyhjennysnopeuteen. Sama on totta myös ohjelmoinnissa. Tyhjennysnopeus voidaan siis approksimoida

$$\frac{[suurin\,tyhjennysalue]}{[alueen\,tyhjennysaika]} = [tyhjennysnopeus]. \tag{1}$$

Approksimaation virhe on pieni, koska huomiotta jätettyjen ohjausbittien suoritusaika on noin kolmesta kuuteen dekadia pienempi kuin huomioitu tyhjennysaika.

Kirjoitussignaali on yleistetyllä tasolla samanlainen kuin tyhjennyskin. Erona on, että nyt ohjausbitit sisältävät lisäksi ohjelmoitavan informaation, joka on kaikilla muistityypeillä yhden sivun kokoinen, kun halutaan suurin tiedonsiirtonopeus. Sivun koko vaihtelee vertailluilla muisteilla 16 tavusta 2 048 tavuun. Kaikki vertaillut muistit pystyvät vähintään jonkinasteiseen rinnakkaisuuteen, joten ajallisesti kaikkien ohjausbittien lähetys vie vain muutamia mikrosekunteja. Varsinainen sivukohtainen kirjoitusoperaatio vie NOReilla millisekunteja ja NANDeilla satoja mikrosekunteja. Aikaero ohjausbittien ja kirjoitusoperaation välillä on kahdesta viiteen dekadia, joten voidaan käyttää approksimaatiota

$$\frac{[suurin\,kirjoitusalue]}{[alueen\,kirjoitusaika]} = [kirjoitusnopeus]. \tag{2}$$

Joillain muisteilla on mahdollista nopeuttaa kirjoittamista syöttämällä suurempaa jännitettä tiettyyn pinniin. Tämä ei vaikuta laskentakaavaan, koska sen ollessa mahdollista, laskussa on käytetty datalehden mainitsemaa kiihdytettyä kirjoitusnopeutta. Kirjoitusnopeuksissa Micronin rinnakkais-NOR [33] erottuu joukosta selvästi muita nopeampana. Datalehden mukaan luku saavutetaan kirjoittamalla haluttu tieto 256 sanan puskuriin. Tarkempia yksityiskohtia datalehdessä ei mainita, mutta puskurin käyttö viittaa siihen, että sanojen kirjoituksessa käytetään rinnakkaisuutta.

Lukeminen noudattaa yleistetysti kuvan 7 esittämää signaalimuotoa. Ohjausbitit sisältävät lukukomennon ja osoitteen sarjamuotoisilla muisteilla ja ainoastaan osoitteen rinnakkaismuotoisilla laitteilla, kun laite on lukutilassa. Sarjamuotoisten NOR-muistien tavallisella lukukomennolla ei ole odotusaikaa, mutta pikalukuoperaatioilla, jotka hyödyntävät rinnakkaisuutta, on muutaman kellojakson viive. Kun sarjamuotoisilla NOR-muisteilla saadaan ensimmäinen luettava tavu piiriltä, muisti voidaan lukea loppuun asti välittämättä mahdollisista sivujen tai lohkojen rajoista. Rinnakkaismuotoisilla muisteilla lukuoperaatio käsittää tavallisesti vain yhden sivun. Sarjamuotoisten NOR-muistien lukunopeus voidaan laskea kaavalla

$$[rinnakkaisia\,ulostuloja] * [suurin\,lukutaajuus] = [lukunopeus], \tag{3}$$

koska maksiminopeus saavutetaan lukemalla koko muistipiiri, jolloin ohjausbitit eivät vaikuta merkittävästi lopputulokseen. Rinnakkaismuotoisilla muisteilla suurin luettava alue on tavallisesti yksi sivu, jossa ensimmäisen tavun satunnaisosoitus kestää

huomattavasti pidempään kuin seuraavien tavujen lukeminen. Suurimman lukualueen ollessa verrattain pieni, lukunopeus lasketaan

$$\frac{[sivukoko]}{[\begin{matrix}satunnais\text{-}\\osoitusaika\end{matrix}] + [\begin{matrix}sivun\,sisäinen\\osoitusaika\end{matrix}] * ([tavuja\,sivussa] - 1)} = [lukunopeus]. \quad (4)$$

Satunnaisosoituksella taulukossa 2 tarkoitetaan aikaa, joka kestää tietyn tavun tai sivun haluamisen ja saamisen välillä. Toisin sanoen mukaan lasketaan ohjausbittien, eli mahdollisen lukukäskyn ja osoitteen sekä sarjamuotoisen lukemisen aiheuttama viive. Rinnakkaismuotoisilla Flash-muisteilla sarjakommunikaation viiveitä ei ole, joten satunnaisosoitus on yksiselitteinen. NOR-muisteilla voidaan osoittaa tavun tarkkuudella, kun NAND-muisteilla ainoastaan sivun tarkkuudella. Kaikilla muistityypeillä satunnaisosoituksen jälkeen samalla sivulla olevien tavujen lukeminen on huomattavasti nopeampaa: NOReilla satunnaisosoitus ja sitä seuraavat luvut kestävät satoja ja kymmeniä nanosekunteja, kun NANDeilla ajat ovat kymmeniä mikrosekunteja ja kymmeniä nanosekunteja samassa järjestyksessä. Kirjallisuudessa toistuvasti mainittu NOR-Flashin tukema suoraajo ilmenee juurikin tässä sekä huomattavasti nopeampina satunnaisosoitusaikoina että tavukohtaisena osoituksena.

Taulukoitujen muistien luotettavuus on valmistajien mukaan hyvin samankaltainen. Jokainen muisti lupaa 100 000 kirjoituksen ja tyhjennyksen muodostamaa jaksoa jokaiselle lohkolle. Toisaalta NANDeillä Eon [36] lupaa 100 000 kierrosta vain, kun käytetään yhden bitin korjaavaa virheenkorjausta puolta kilotavua kohti. Myös Micron [37] ja Toshiba [39] suosittelevat samanlaajuista virheenkorjausta ja Spansioniin [38] kyseinen virheenkorjaus on integroitu. Jokainen NOR-muisti mainostaa, että tieto säilyy 20 vuotta ja jokainen NAND-muisti, joka mainitsee säilyvyyden, mainitsee arvoksi 10 vuotta. Toisaalta Toshiban [39] datalehdessä nämä arvot on esitetty kuvaajana, jossa tiedon säilyvyys heikkenee uudelleenkirjoitusten kasvaessa. Luotettavuuteen palataan luvussa 3.4.

Suurin osa vertailluista muisteista noudattaa jonkinlaista rajapintastandardia ja noudattaa yhtenevää käytäntöä muistin tilatiedon ilmoittamisessa. Erityisesti NOR-muistien käyttämät rajapinnat määrittävät ainoastaan, kuinka muistit ilmoittavat omat ominaisuutensa ja sisäisen rakenteensa [40][41]. Vertailu ei anna todellista kuvaa NANDien rajapinnoista, koska erilaisia rajapintoja ja standardeja on paljon erilaisia: muutama yleinen standardi ja monia valmistajakohtaisia. Rajapintoihin palataan tarkemmin luvussa 3.5. Kirjoituksen tai tyhjennyksen käynnissäolo ilmoitetaan tavallisesti siten, että sarjamuotoiset muistit sisältävät rekisterin, joka kertoo muistin tilan, ja rinnakkaismuotoiset muistit sisältävät joko rekisterin tai sähköisen tilatietopinnin.

Muistipiirin hinta on huomattavasti muita taulukon 2 arvoja epämääräisempi, mutta hyvin olennainen tekijä kustannustehokasta muistiratkaisua etsivälle elektroniikkasuunnittelijalle. Epämääräisyys johtuu esimerkiksi siitä, että markkinahinta voi vaihdella nopeasti ja ostamalla useita muistipiirejä, hinta voi olla edullisempi kuin jos niitä ostaa vain muutaman. Taulukkoon valittu hinta on alin Octopart-elektroniikkakomponenttihakukoneella 6.8.2013 löydetty. Pienin hankintamäärä tämän hinnan saavuttamiseksi vaihteli paljon eri komponenteilla, joten myös hankintamäärät löytyvät taulukosta. Lisäksi muistivalmistajalla ja muistipiirejä käyttävällä yrityksellä

voi olla kahdenkeskeisiä sopimuksia, joiden ansiosta muistipiirin hinta on vieläkin edullisempi. Taulukoitujen hintatietojen perusteella nähdään, että jos sovellukselle riittää 64 Mb tallennuskapasiteetti, NOR-muisti voi hyvinkin olla edullisempi kuin pienimmät saatavilla olevat NAND-muistit. Tilanne saattaa kuitenkin olla erilainen jos tarvitaan suurempaa tallennuskapasiteettia. On myös syytä muistaa, että vertailu ei sisällä usean suurimman NAND-valmistajan muisteja, koska niiden datalehtiä ei ollut saatavilla, joten on mahdollista, että on saatavilla esitettyä edullisempia NAND-piirejä.

Taulukkoon 3 on koottu taulukon 2 muistien kirjoitus- ja tyhjennysajat sekä muistiyksikön koko, jota operaatio koskee. NANDeillä muistialueiden koot on ilmaistu muodossa $X + Y$, jossa X tarkoittaa varsinaiselle tiedolle varattua aluetta ja Y virheenkorjauskoodille sekä muille järjestelmätiedoille varattua aluetta. Muistit ovat edelleen jakaantuneet selkeästi kolmeen kategoriaan: sarjamuotoista rajapintaa käyttävät NORit, rinnakkaisen rajapinnan NORit sekä NANDit. Sarjamuotoiset NORit voidaan tyhjentää lohkojen lisäksi myös alilohkoina niiden datalehtien mukaan. Toisaalta perinteisen Flash-terminologian mukaan pienin tyhjennettävä yksikkö on lohko, joten vaihtoehtoinen näkökulma on, että sarjamuotoisten NORien lohkokoko on pienempi kuin muilla vertailun muisteilla. NOR-muistit voivat kirjoittaa yksittäisiä tavuja, tai sanoja [35], mikä on yksi NORien erityisominaisuuksista NANDiin verrattuna. Silti kaikki datalehdet eivät anna suorituskykyarvoja operaatiolle. Laskennallisesti parempi suorituskyky saavutetaan kirjoittamalla sivun kokoinen puskuri kerralla. Sivukohtaiset kirjoitusajat on siten myös listattu taulukkoon. Monilla muisteista on kirjoituksen kiihdytysominaisuus, mutta sen suorituskykyä ei ole merkitty yhtenäisesti datalehtiin, joten sitä ei huomioitu tässä taulukossa. Niissä datalehdissä joihin sen vaikutus oli selkeästi merkitty, kiihdytys pienensi tavallisesti kirjoitusajan tyypillistä arvoa 10–20 %.

Muistien kirjoituksen- ja tyhjennyksen suorituskyvystä todetaan taulukon 3 perusteella, että samantyyppisten muistien – NORien tai NANDien – välillä arvot ovat hyvin samankaltaisia, mutta erityyppisten muistien välillä erot ovat suuria. Yksittäisen muistin kirjoitus- tai tyhjennysajan tyypillisen ja maksimiarvon välillä on suuri ero. Useissa datalehdissä mainitaan, että tyypilliset arvot pätevät huoneenlämmössä, kun käyttöjännite on sallittujen rajojen keskellä ja muistilohkoja on tyhjennetty vain vähän. Maksimiarvot taas ovat voimassa, kun lämpötila, käyttöjännite tai tyhjennyskertojen lukumäärä on lähellä sallittuja raja-arvoja. Toisaalta on myös datalehtiä, jotka painottavat ainoastaan tyhjennyskertojen lukumäärää perustellessaan maksimiarvoja [29]. Tämä viittaa siihen, että kirjoitusten ja tyhjennysten lukumäärällä on suuri vaikutus muistien operaatioiden hidastumisessa. Luvussa 3.4 käsitellään tarkemmin tekijöitä, jotka vaikuttavat tähän. Teoriassa mainittiin, että NOR-muistien kirjoitusmenetelmä on NANDien käyttämää nopeampi. Kun verrataan molempien muistityyppien pienimpien kirjoitusalojen ohjelmointiaikoja, tämä pitää paikkansa. Jos valitaan jokin yhteinen kirjoitusala, esimerkiksi sivu, NAND on nopeampi, koska se kirjoittaa tiedon soluihin rinnakkain. Molemmat muistityypit käyttävät samaa tyhjennysmenetelmää ainoastaan pienin eroin, joten on mielenkiintoista huomata valtava suorituskykyero tyhjennysajassa, vaikka vertaisi sarjamuotoisilla NOReilla alilohkoa ja NANDeillä tavallista lohkoa keskenään. Tähän liittyvät yksityiskohdat ovat epäselvät. Muistetaan kuitenkin, että NOR-soluilla on paksumpi oksidikerros, joka vaikuttaa FN-tunneloinnin suorituskykyyn.

Datalehdissä painotettiin oikean käyttöjännitteen lisäksi oikeaa käynnistys- ja

Taulukko 3: Markkinoilla olevien Flash-muistien absoluuttiset kirjoitus- ja tyhjennysajat tietoaloittain

	1. Eon Silicon Solution EN25QH64	2. Macronix International MX25L6435E	3. Micron M25PX64	4. Micron N25Q064A	5. Spansion S25FL064P	6. Winbond W25Q64FV	7. Adesto Technologies AT45DB642D	8. Eon Silicon Solution EN29GL064	9. Macronix International MX29LV640E	10. Micron PC28F064M29EWXX	11. Spansion S29GL064N	12. Winbond W29GL064C	13. Eon Silicon Solution EN27LN1G08	14. Micron NAND128-A	15. Spansion S34ML08G1	16. Toshiba TC58DVG02D5TA00
Koko piirin koko [Mt]	8	8	8	8	8	8	8	8	8	8	8	8	128	16	128	128
→ tyhjennysaika, tyyp. [s]	30	50	68	60	64	20	-	16	45	66	64	19	-	-	-	-
→ tyhjennysaika, maks. [s]	70	80	160	120	128	100	-	60	65	262	128	128	-	-	-	-
Lohkokoko[1] [kt]	64	64	64	64	64	64	256	64	64	64	64	64	128+4	16+0,5	128+4	128+4
→ tyhjennysaika, tyyp. [s]	0,3	0,7	0,7	0,7	0,5	0,15	0,7	0,1	0,5	0,5	0,5	0,15	0,002	0,002	0,004	0,003
→ tyhjennysaika, maks. [s]	2	2	3	3	2	2	1,3	2	2	4	3,5	2	0,010	0,003	0,010	0,010
Alilohkokoko[1] [kt]	4	4	4	4	4	4	1	-	-	-	-	-	-	-	-	-
→ tyhjennysaika, tyyp. [s]	0,06	0,06	0,07	0,25	0,20	0,06	0,02	-	-	-	-	-	-	-	-	-
→ tyhjennysaika, maks. [s]	0,30	0,30	0,15	0,80	0,80	0,40	0,04	-	-	-	-	-	-	-	-	-
Sivukoko [t]	256	256	256	256	256	256	1056	16	16	16	16	16	2k+64	512+16	2k+64	2k+64
→ kirjoitusaika, tyyp. [ms]	1,3	1,4	0,8	0,5	1,5	0,7	3	0,008	0,011	0,015	0,060	0,006	0,2	0,2	0,2	0,3
→ kirjoitusaika, maks. [ms]	5	5	5	5	3	3	6	0,200	0,360	0,180	0,480	0,200	0,7	0,5	0,7	0,7
Tavun kirjoitusaika, tyyp. [us]	-	12	25	15	-	20	-	8	9	15	-	6[2]	-	-	-	-
Tavun kirjoitusaika, maks. [us]	-	300	5000	5000	2048	50	-	200	300	175	-	200[2]	-	-	-	-

[1]Osassa komponentteja koko muistia pienempiä, mutta sivua suurempia tietoyksikköjä on useita. Tässä lohko tarkoittaa datalehdessä mainittua lohkoa tai sektoria. Osalla muisteista pienin tyhjennettävä yksikkö oli perinteisestä termistöstä poiketen alilohko tai alisektori. Alilohkokohtaiset suorituskyvyt on listattu erikseen.

[2]Arvo on sanan kirjoitusaika, koska kyseinen muisti ei tue yksittäisen tavun kirjoittamista. Molempia tukevien muistien datalehtien perusteella tavun ja sanan kirjoitusajat eroavat ainoastaan muutamilla mikrosekunneilla.

sammutusmenettelyä. Aiemmin luvussa 3.2 mainittiin, että datalehtien mukaan käyttöjännitteen ollessa sallittujen arvojen rajoilla, kirjoitus- ja tyhjennysajat voivat olla pidempiä verrattuna siihen, että käyttöjännite on sallittujen rajojen keskellä. Lisäksi käytännössä kaikissa datalehdissä korostettiin, että kun muistiin tuodaan käyttöjännite, sillä kestää kymmeniä mikrosekunteja käynnistyä. Tänä aikana muut kuin muistin tilaa tiedustelevat komennot voivat epäonnistua. Jotkin datalehdet ohjeistivat myös toimista muistia sammutettaessa: kun käyttöjännite laskee, eräät pinnit tulee asettaa tiettyyn tilaan ja muistille ei saa antaa uusia käskyjä. Tseng [42] on tutkinut Flash-muistin käyttäytymistä, kun sähkö katkeaa yllättäen kesken muistin käytön. Lyhyesti hänen tuloksensa osoittivat, että juuri kirjoitettaessa olevan tiedon katoamisen lisäksi sähkökatko vaikuttaa myös solun luotettavuuteen jatkossa. On siis mahdollista, että vaikka sähkökatko ei aiheuttaisi välittömästi suurta vahinkoa, paljon myöhemmin kyseinen lohko on muita herkempi lukuhäiriöille ja tieto säilyy siinä lyhyempään.

3.4 Luotettavuus

Kaikki elektroniikkakomponentit hajoavat joskus ja Flash-muisteilla on tässä suhteessa erityisen huono maine, sillä tyhjennyskertojen rajallisuus on yleisesti tiedossa. Luotettavuus on poikkeuksellisen tärkeää, kun suunnitellaan laitetta, jonka käyttöikä on hyvin pitkä. Kirjallisuudesta löytyy paljon erilaisia Flashin luotettavuusongelmia ja vikaantumismekanismeja, mutta monet näistä ovat asioita, joihin ainoastaan muistipiirin valmistaja voi vaikuttaa. Muistivalmistajan näkökulma ei ole tämän työn keskiössä, mutta mikäli lukija on kiinnostunut, Brewer [9, s. 445–590] antaa laajan alustuksen aiheeseen liittyvään tutkimukseen. Alan kirjallisuuden ja julkaisujen painotuksien perusteella voidaan kuitenkin olettaa, että käsiteltävät luotettavuusaiheet kattavat valtaosan käytännössä ilmenevistä vikaantumisista. Kyseiseen huomioon palataan vielä kappaleen loppupuolella, kun tarkastellaan vikaantumismekanismien todennäköisyyksiä. Tässä luvussa selvitetään, millä tavoin Flash-muistit vikaantuvat ja miten vikaantumistodennäköisyyttä voidaan pienentää.

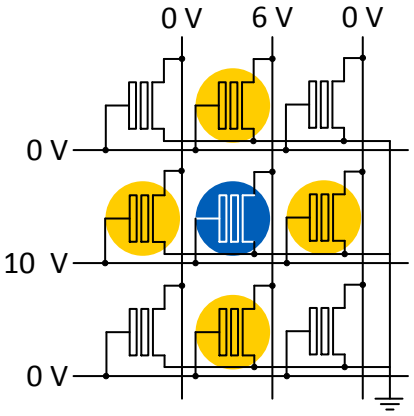
Flash-muistien tyhjennyskertojen rajallisuus johtuu solun kelluvaa hilaa ympäröivän *oksidin pilaantumisesta (engl. oxide degradation)*. Tyhjennyskerroilla tarkoitetaan lohkon tyhjennyksen ja sen kirjoittamisen muodostamaa kierrosta, jota voidaan kutsua myös *kirjoitus- ja tyhjennyssykliksi (engl. P/E cycle, program/erase cycle)*. Muistipiirin datalehteen kirjattu tyhjennyskertojen minimimäärä on lohkokohtainen, joten vaikka yksi lohko kulutetaan loppuun, muut lohkot ovat edelleen täysin kunnossa. Kirjoitus- ja tyhjennysoperaatiot altistavat muistisolun lukuoperaatiota korkeammalle sähkökentälle, mikä aiheuttaa oksidin pilaantumista. NAND-muistit kuluttavat muistisoluja NOReja enemmän, koska ne käyttävät sekä kirjoittamiseen että tyhjentämiseen FN-tunnelointia, joka vaatii suuremman sähkökentän kuin CHE-injektointi, jota NOR-muistit käyttävät kirjoittamiseen. Brewer [9, s. 447–461] selittää yksityiskohtaisemmin, että ohjelmoitaessa ja tyhjennettäessä osa elektroneista tarrautuu oksidiin sekä oksidin ja kanavan rajapintaan, mitä kutsutaan *elektronien tarrautumiseksi (engl. electron trapping)*. Ylimääräiset elektronit muuttavat muistisolun kynnysjännitettä, mikä vaikuttaa lukuoperaation lopputulokseen ja siten myös kirjoituksen tai tyhjennyksen onnistumisen tarkistuksen. Tarrautuneet elektronit myös heikentävät kanavan transkonduktanssia, mikä vaikuttaa

kirjoitus- ja tyhjennysoperaatioon hidastavasti. Toisaalta korkea lämpötila voi irrottaa tarrautuneita elektroneja, joka taas muuttaa kynnysjännitettä päinvastaiseen suuntaan. Parempia oksidimateriaaleja on kehitetty ja kehitteillä, mutta nämä ovat muistivalmistajan vastuulla olevia asioita, joihin piiriä sovelluksessaan käyttävä suunnittelija ei voi vaikuttaa. Sen sijaan suunnittelija voi käyttää *kulutuksentasausalgoritmia (engl. wear leveling)*, jolla tiettyyn muistilohkoon kohdistuva kulutus jaetaan koko muistipiirin alueelle [21]. Kulutuksentasaus liittyy läheisesti *muistiohjaimeen (engl. memory controller)*, johon palataan myöhemmin tässä luvussa.

Tieto säilyy Flash-muistilla vain rajallisen ajan, jos sitä ei muokata, mutta käytännössä ongelma ei ole suuri. Haihtumaton muisti määritellään usein siten, että tieto säilyy muistilla vähintään kymmenen vuotta, jos sitä ei käytetä [7]. Edellisessä luvussa vertaillut NOR-muistien datalehdet ilmoittavat tiedon säilyvyydeksi 20 vuotta ja NAND-muistit kymmenen vuotta, jonka jälkeen tieto saattaa muuttua muistilla ilman ulkoisia toimia. Brewer [9] selittää, tämän johtuvan siitä, että kelluvaa hilaa ympäröivä dielektrinen materiaali tai hilan oksidikerros vuotaa elektroneja. Vuodon suunnasta riippuen kelluvan hilan varaus laskee tai kasvaa, mikä muuttaa solun arvon toiseksi ajan myötä. Säilyvyysarvon ollessa vuosia, se lasketaan uusille muistipiireille kiihdytetyllä elinikätestillä, jossa piiriä säilytetään korkeassa lämpötilassa ja sen perusteella ennustetaan elinikä matalammassa lämpötilassa. Texas Instrumentsin julkaisuissa [43][44] on osoitettu, että kiihdytetyn elinikätestin perusteella muistin elinikä voi olla moninkertainen, jopa satakertainen, huoneenlämmössä (20 °C) verrattuna muistipiirien maksimilämpötilaan ($\geq$ 85 °C). Taulukon 2 muistien datalehdissä piirin elinikä ilmoitettiin useimmiten 55 °C lämpötilassa tai lämpötilaa ei ollut mainittu. Ratkaisuksi tilanteisiin, jossa tarvitaan pidempi muistin elinikä, Texas Instruments suosittelee muistin virkistystä, eli että muisti tyhjennetään ja uudelleenkirjoitetaan muistille luvatun eliniän välein. Koska elinikä johtuu hiljalleen vuotavista muistisoluista, muistin säilyvyys voidaan tällä tavalla palauttaa alkutilaansa aina uudestaan. Lisäksi sama lähde suosittelee tekemään säännöllisen eheyden tarkistuksen esimerkiksi jollakin tiivistealgoritmilla. Toshiba [39] esittää datalehdessään, että Flash-muistin tiedon säilyvyys laskee, jos tietoa uudelleenkirjoitetaan usein[13], joten muistin virkistys ei toimi teoriassa rajattoman pitkään. Cai [45] on tutkinut tätä näkökulmaa muutama tuhat tyhjennyssykliä kestävillä monitasoisilla NAND-Flasheillä ja esittää, että virhekorjaavalla virkistyksellä, jonka suoritustaajuutta nostetaan kirjoituskertojen lukumäärän kasvaessa, voidaan pidentää tiedon säilyvyys käytännössä yli 40-kertaiseksi. Mainitsemisen arvoista hänen algoritmissaan on, että se voidaan toteuttaa ohjelmistolla täysin ilman laitteistomuutoksia.

Muistisolu voi kärsiä *luku-, kirjoitus ja tyhjennyshäiriöstä (engl. read, write, and erase disturb)*, kun sen lähellä olevalle solulle tehdään jokin näistä operaatioista. Jos muistirakenne, olkoon kyseessä NAND tai NOR, ajatellaan matriisina, jossa pystysuunnassa kulkevat bittilinja ja vaakasuunnassa sanalinjat, luettaessa tietyn solun arvo, aktivoidaan yksi sanalinja ja bittilinja. Lukuoperaatio aiheuttaa jännitteen myös muistisolujen yli, jotka kuuluvat samaan sanalinjaan kuin luettava solu, mutta joiden bittilinjaa ei aktivoitu eikä toisin sanoen haluta lukea. Jos tämä ylimääräinen

[13] Kuvaajassa ei ole numeerista asteikkoa, joten tarkkoja lukuarvoja ei voida mainita.



Kuva 8: Kirjoitushäiriö NOR-tyyppisessä muistissa. Kuvassa keskellä on tarkoituksella kirjoitettava solu, jonka ympärillä on useita tahattomasti kuormittuvia soluja. [9, 491–495]

sähkökentälle altistaminen aiheuttaa muutoksen solun tilassa, esimerkiksi tyhjä solu tulkitaan ohjelmoiduksi, sitä kutsutaan *lukuhäiriöksi (engl. read disturb)*. NAND-muistit ovat alttiimpia tälle häiriölle, koska niiden lukuoperaatio kestää kymmeniä mikrosekunteja [8, s. 502-503], kun vastaavasti NOReilla lukuoperaatio kestää alle yhden mikrosekunnin [9, s. 490]. Mielenkiintoista on huomata, että muistilohkon tyhjennyskertojen lukumäärä ja siellä aiheutuvien lukuhäiriöiden suhde noudattaa potenssilakia. Toisin sanoen, kun lohkoa on kirjoitettu ja tyhjennetty monta kertaa, se on alttiimpi lukuhäiriöille. [46] Läheisesti lukuhäiriöihin liittyy myös *ohitushäiriö (engl. pass disturb)*, joka ilmenee vain NAND-tyypillä, jolla samalla bittilinjalla olevien muistisolujen yli on suurempi jännite kuin luettavan solun ylitse. Ohitushäiriöllä tarkoitetaan niiden solujen tilan tahatonta muuttumista, jotka ovat samassa bittilinjassa luettavan solun kanssa. *Kirjoitushäiriö (engl. write disturb)* on hieman monimutkaisempi. Se vaikuttaa soluihin, jotka jakavat bittilinjan tai sanalinjan ohjelmoitavan solun kanssa. Kuvassa 8 on havainnollistettu tätä. Kirjoitusoperaatio saattaa tahattomasti tyhjentää aiemmin ohjelmoituja soluja tai ohjelmoida tyhjiä soluja, mutta erityisesti se ilmenee solujen varausjakauman leventymisenä. Tässäkin tapauksessa kirjoitusaika vaikuttaa häiriön ilmenemisen todennäköisyyteen, mutta toisaalta kirjoitusoperaatio suoritetaan harvemmin kuin lukuoperaatio. *Tyhjennyshäiriöt (engl. erase disturb)* ovat luonnollinen jatkumo häiriöille, mutta koska tyhjennysoperaatio kohdistetaan lohkoille, jotka on eristetty toisistaan, se ei aiheuta käytännössä ongelmia. Luku- ja kirjoitushäiriöt aiheuttavat ongelmia erityisesti monitasoisilla soluilla, joissa varausjakaumilla on pienemmät virhemarginaalit kuin yksitasoisilla soluilla [8, s. 98].

NAND-muisteilla on useampia ja ongelmallisempia virhelähteitä kuin NOR-muisteilla, minkä takia erityisesti NAND-muistien kanssa suositellaan *virheenkorjauskoodeja (ECC, engl. Error Correction Codes)* käyttöä. Edellä käsiteltiin erilaisia häiriöitä, jotka vaikuttavat molempiin muistityyppeihin, mutta vähemmän NORiin sen rakenteen vuoksi. Samoin edellä käsitelty NANDin käyttämän kirjoitusmenetelmän vaatima NORia suurempi sähkökenttä nopeuttaa muistisolun oksidin kulumista, mikä heikentää luotettavuutta entisestään. NAND-solut voidaan valmistaa NOR-soluja pienemmiksi, mi-

kä yhtäältä tarkoittaa parempaa skaalattavuutta, mutta toisaalta vaatii monimutkaisemman valmistusprosessin, jolla ei voida taata, että kaikki solut ovat täysin toimintakykyisiä [9, s. 23]. Perinteisesti valmistusteknisiä ongelmia vastaan on kamppailtu lisäämällä muistipiirille toistoa, eli valmistettu ylimääräisiä muistisoluja sekä oheiselektroniikkaa, ja piirin valmistuksen loppuvaiheessa vialliset ominaisuudet on kytketty pois päältä jättäen ainoastaan riittävä määrä toimivia osia [8, s. 353–392]. Valmistusaikaista toistoa käytetään edelleen, mutta nykyään lisäksi käytetään virheenkorjausta. Virheenkorjaus on matemaattinen menetelmä, jossa varsinaisen tiedon lisäksi tallennetaan joukko ylimääräisiä bittejä, joiden avulla voidaan päätellä, onko joku (tai useampi) tallennetuista biteistä muuttunut itsestään ja mikä bitti on kyseessä. Tämän tiedon perusteella bittivirheet voidaan korjata tiettyyn rajaan asti. Moon [47] on kirjoittanut virheenkorjauskoodien teoriasta yleisesti ja Stievano [48, s. 31–82] sekä Micheloni [8, s. 393–422] kertovat niistä Flash-muistien näkökulmasta. Virheenkorjauskoodeja ei kannata kasvattaa kuitenkaan loputtomasti, sillä koodien vaatima muistipiirin pinta-ala ja tarvittu sähköteho kasvavat nopeammin kuin korjattujen bittien lukumäärä [45]. Edellisen luvun muistivertailun yksitasoisille NAND-muisteille suositeltiin tai ne sisälsivät yhden bitin virheenkorjausta puolta kilotavua kohti.

Kulutuksentasaus, viallisten lohkojen hallinta ja virheenkorjaus ovat usein erillisen muistiohjaimen tehtäviä. Micheloni [21] kuvailee sen rakennetta ja tärkeimpiä tehtäviä. Muistiohjain on sovelluskohtainen suoritinpiiri, jonka päätoiminnot on toteutettu *laiteohjelmalla (engl. firmware)* tai ajoituskriittiset osat suoraan *laitteistotasolla (engl. hardware)*. Ohjainpiiri erottaa muistiin kirjoitettavan tiedon loogisen ja fyysisen sijainnin: looginen sijainti on isäntälaitteen näkökulma muistin rakenteesta ja fyysinen sijainti tarkoittaa muistibittien varsinaista sijaintia muistipiirillä. Kulutuksentasaus toimii siten, että kun lohkoa pienempi osa muistista halutaan päivittää, tiedon vanha sijainti merkitään ”vanhaksi” ja päivitettävä tieto kirjoitetaan loogisesti samaan paikkaan, mutta fyysisesti muualle. Koska Flash-muisteissa kirjoitus ja tyhjennysoperaatiot kohdistuvat eri kokoisille alueille, tämä menetelmä vähentää tyhjennyskertoja ja toisaalta estää tilanteen, jossa osa muistilohkoista on huomattavasti keskimääräistä kuluneempia. ”Vanhoiksi” merkittyjen alueiden uudelleenkäyttöönottoa kutsutaan *roskienkeräykseksi (engl. garbage collection)*. Operaatio voidaan suorittaa taustalla, jolloin se ei vaikuta muistin suorituskykyyn. Erityisesti NAND-muisteilla on jo tehtaalta valmistuessaan niin sanottuja *viallisia lohkoja (engl. bad block)*, eli lohkoja joiden luotettavuutta ei voida taata, mutta niitä syntyy myös muistin käytön aikana lohkon solujen kuluessa. Muistiohjain pitää kirjaa niistä ja estää niiden käytön tarvittaessa. Virheenkorjausta käsiteltiin aiemmin, mutta se on myös yksi muistikontrollerin päätehtävistä. Michelonin julkaisu koskee erityisesti muistikortteja, joten laajemmissa muistipiirejä sisältävissä sovelluksissa muistiohjain voi olla osa pääsuoritinta tai *kenttäohjelmoitavaa porttimatriisia (FPGA, engl. Field-Programmable Gate Array)*.

Muistisolujen varausjakauman tiukempien marginaalien vuoksi monitasoiset muistit kärsivät muistisolujen tahattomasta varauksen muuttumisesta – johon tyhjennyskertojen lukumäärä, tiedon säilyvyys ja eri operaatioiden häiriöt liittyvät hyvin läheisesti – ja ovat luotettavuusmielessä huomattavasti vaikeampia kuin yksisoluisia muistisoluja käyttävät piirit. Kuvassa 6 havainnollistettiin varausjakaumia ja niiden tiukkenemista, kun soluun tallennetaan useampia bittejä. Taulukosta 2 nähtiin, että valmistajat lupaavat yksisoluisille eli kahden varausjakauman muisteille yleisesti 100 000

uudelleenkirjoituskertaa. Kaksitasoisille eli neljän varausjakauman muisteille luvataan enää muutama kymmenen tuhatta tyhjennyskertaa [49][21]. Varausjakaumien lukumäärän noustessa kahdeksaan, kolmitasoisten muistisolujen bittivirheiden määrä nousee vain tuhansien tyhjennyskertojen jälkeen tasolle, jossa vaaditaan jo huomattavan pitkiä virheenkorjauskoodeja luotettavuuden takaamiseksi [45][50].

Flash-muistikortit ja niin kutsutut muistitikut sisältävät muistipiirin ja -ohjaimen. Muistikorttien ja -tikkujen rakenne on samankaltainen, ainoastaan rajapinta on erilainen. Ne koostuvat muistiohjaimesta, varsinaisista muistipiireistä, joita voi olla yksi tai useampia, sekä tarpeellisesta oheiselektroniikasta. Micheloni kertoo muistikorteista julkaisussaan [21] ja kirjassaan[14] [8, s. 483–514]. Tekstien perusteella irrotettavien muistien ominaisuudet sähköisessä ja luotettavuusmielessä ovat identtiset erillisten Flash-muistipiirien ominaisuuksiin. Suurin ero on sisäänrakennettu muistiohjain.

Flash-muisteilla on lukuisia vikaantumismekanismeja, jotka erikseen käsiteltynä voivat antaa totuutta kielteisemmän kuvan muistien luotettavuudesta. Sun [49] ja Cai [51] ovat aivan viime vuosina tutkineet kaksitasoisten NAND-piirien päävikaantumismenetelmiä, eli tiedon säilyvyyttä, luku- ja kirjoitushäiriöitä ja lisäksi tyhjennysoperaatioiden epäonnistumista. Viimeisintä näistä ei ole käsitelty tässä työssä, mutta se tarkoittaa, ettei tyhjennysoperaatio tyhjennä solua. Tyhjennysvirhe voi johtua valmistusvirheestä tai solun oksidiin tarrautuneista elektroneista, eli muistisolun pilaantumisesta. Molemmissa julkaisuissa havaittiin, että kaikki virhemekanismit voimistuvat eksponentiaalisesti tyhjennyskertojen funktiona. Lisäksi bittivirheiden todennäköisyys kasvoi 20 000 tyhjennyskerran aikana useita dekadeja monilla mekanismeista. Ainoastaan jälkimmäinen tutkimus asettaa virhelähteet selkeästi järjestykseen, mutta sen tulosten perusteella tiedon säilyminen on selkeästi hallitseva vikaantumismenetelmä. Tiedon säilyvyyttä tutkittiin usealla eri aikavälillä, mutta 1 000 tyhjennyskerran jälkeen se dominoi kaikkia muita virhelähteitä, pahimmillaan useita dekadeja todennäköisempänä. Tutkimuksessa ei kuitenkaan ollut käytetty muistin virkistämistä, jolla on saatu hyviä tuloksia muissa tutkimuksissa [45].

Tässä luvussa tutustuttiin osaan Flash-muistien lukuisista vikaantumismenetelmistä. Aiheesta kirjoitettu tutkimus antaa kuitenkin ymmärtää muutaman mekanismin kattavan suurimman osan Flash-muistien vikaantumisista, tiedon säilyvyyden ollessa ongelmallisin. Kaikkiin näistä on kuitenkin kehitetty menetelmiä, joilla vikaantumista voidaan rajoittaa, usein jopa tehokkaasti. Lisäksi kaikkien suurimpien vikaantumismekanismien ollessa verrannollisia tyhjennyskertojen lukumäärän aiheuttamaan muistisolujen oksidin pilaantumiseen, ne korostuvat erityisesti monitasoisilla Flash-muisteilla, jossa varausjakaumat ovat lähempänä toisiaan.

3.5 Rajapinnat

Aiemmissa luvuissa on käsitelty Flash-muisteja pääasiassa elektroniikan kannalta, mutta niihin liittyy olennaisesti myös ohjelmistonäkökulma, koska jotta muistista on hyötyä, sen käyttö vaatii ajurin. Ajurikehityksen vaatimien resurssien ja ohjelmistovirheiden minimoimiseksi on toivottavaa, ettei jokaiselle käytettävälle muistille – ja niiden vaihtoehtoisille komponenteille – tarvitse kehittää erilaista ajuria. Tässä luvussa

[14] Tarkemmin, kyseinen luku on muistivalmistaja Micronin A. Ghilardellin ja S. Cornon kirjoittama.

tutustutaan NAND- ja NOR-muistien käyttämiin rajapintastandardeihin ja selvitetään, onko mahdollista kehittää yksi ajuri, jota voidaan käyttää usean muistipiirin kanssa.

Taulukon 2 muistien datalehdistä nähdään, että rinnakkaistyyppiset NOR-muistit jakavat yhtenäisen käskykannan ja yhtä lukuun ottamatta sarjatyyppisten NORien käskykanta on yhtenäinen niiltä ominaisuuksilta, joita ne tukevat. Kaikki sarja-NORit eivät tue täsmälleen samoja toimintoja, esimerkiksi lukuoperaatiota neljällä rinnakkaisella signaalilla tai tietynkokoisen alilohkon tyhjennystä. Perusominaisuudet, kuten luku-, kirjoitus- ja tietynkokoinen tyhjennysoperaatio ovat tuettu kaikissa sarja-NOReissa. Poikkeus lähes kaikkeen tässä luvussa sarjatyyppisistä NOReista kerrottuun on Adeston muistipiiri [30]. Valmistajan mukaan kyseinen piiri perustuu NOR-teknologiaan, mutta siitä käytetään myös nimeä DataFlash, koska se käyttää alunperin Atmelin kehittämää yksityistä rajapintaa. Esimerkiksi elektroniikkakomponenttien jälleenmyyjät Mouser ja Digi-Key luokittelevat DataFlashin muuksi kuin NOR- tai NAND-muistiksi. Tieto muistia muokkaavien operaatioiden käynnissäolotilasta on yleensä talletettu muistin tilarekisteriin, mikä vaatii, että ajurin tulee kysellä tietoa säännöllisin väliajoin. Rinnakkaismuotoisilla NOReilla tilatieto on saatavilla tavallisesti pinnin kautta. Erillisen pinnin avulla tilatiedon vaihtumiseen voidaan reagoida keskeytyksellä ilman erillistä kyselyä. Jotkut NOR-muisteista poikkeavat näistä pääsäännöistä siten, että ne sisältävät molemmat tilatiedot: tilarekisterin sekä tilapinnin. Lyhyesti voidaan todeta, että yksittäistä poikkeusta lukuun ottamatta, taulukon 2 NOR-muistit voidaan jakaa sarja- ja rinnakkaismuotoisiin, joiden molempien sisältä löytyy joukko ryhmälle yhteisiä komentoja samoilla *käskykoodeilla (engl. opcode, operation code)*, joilla voidaan käyttää muistin perustoimintoja, eli luku-, kirjoitus- ja tyhjennysoperaatioita.

Käskykannan ja tilatiedon lisäksi NOR-muistien rajapintaan vaikuttavat *Elektronilaitetekniikan yhteisneuvoston (JEDEC, engl. Joint Electron Devices Engineering Council)* kaksi standardia: JESD68.01 ja JESD216. Vanhempi näistä on JESD68.01, eli *Yleinen Flash-rajapinta (CFI, engl. Common Flash Interface)*, alun perin vuodelta 1999. Se määrittelee yhtenäisen tavan isäntäjärjestelmälle tiedustella Flash-piirin parametreja valmistajasta ja tarkasta muistipiiristä riippumatta. Sitä käyttäviltä muistipiireiltä saadaan neljää erityyppistä tietoa: piirin käyttämien jännitteiden rajat, eri operaatioiden tyypilliset ja maksimikestot, muistipiirin geometria[15] ja valmistajakohtaiset merkinnät. [40] Standardissa huomattavaa on, että se määrittää kyselytavan vain muutamien perustoimintojen kestoille. Jos siis muistipiiri tukisi muitakin komentoja, niiden suoritusaikatiedot eivät sisältyisi CFI:hin. Kaikki taulukon 2 rinnakkaismuotoiset NOR-Flashit noudattavat CFI-standardia. Uudempi JEDEC-standardi on JESD216, eli *Sarjamuotoisten Flashien kyseltävät parametrit (SFDP, engl. Serial Flash Discoverable Parameters)*, vuodelta 2011. Se on tarkoitettu nimenomaan sarja-NOReille ja sisältää tiedon piirin muistigeometriasta sekä siitä, että mitä lukuisista komennoista muisti tukee. Lisäksi standardi antaa valmistajille mahdollisuuden lisätä omia parametrejaan mukaan. [41] SFDP eroaa CFI:stä erityisesti siinä, ettei se sisällä piirin käyttöjännitteitä tai komentojen aikainformaatiota. Lisäksi SFDP luettelee useita operaatioita, jotka sarja-NORit voivat sisältää, mikäli valmistaja niin haluaa. Mielenkiintoisesti ainakaan taulukossa 2 vertailluista muisteista mi-

[15]Muistin geometrialla tarkoitetaan CFI-standardissa tietoa siitä, minkä kokoisiin muistialueisiin tieto on talletettu ja kuinka monta minkäkin kokoisia alueita on. Tässä tekstissä samaa termiä käytetään myös muihin standardeihin liittyen tarkoittamaan samaa asiaa.

kään ei noudata molempia CFI- ja SFDP-standardeista. Joka tapauksessa JEDECin kaksi erityisesti NORien käyttämää Flash-rajapintastandardia helpottavat ajurikehitystä, koska järjestelmän ei tarvitse tietää tarkalleen siinä olevaa muistipiiriä, vaan se voi mukautua eri kokoiseen ja jossain määrin eri operaatioita tukeviin muistipiireihin automaattisesti ilman ohjelmakoodimuutoksia.

Taulukko 4: Erilaisia NAND-rajapintoja [52]

	Puhdas NAND	Valmistajien standardit	JEDEC e·MMC	JEDEC UFS
Monitasoisuus	1	≥ 1	> 1	> 1
Rajapinta	Rinnakkainen	?	Rinnakkainen	Sarja
Nopeus [Mt/s]	50–400	?	200	750
Virheenkorjaus	Ei	?	Kyllä	Kyllä
Kulutuksentasaus	Ei	?	Kyllä	Kyllä
Viallisten lohkojen hallinta	Ei	?	Kyllä	Kyllä
Roskienkeräys	Ei	?	Kyllä	Kyllä
Kapasiteetti [Gt]	0,064–16	?	2–128	16–128
Virheenkorjaus isäntäjärjestelmässä [b]	1–24+	?	0	0

NAND-muisteilla valmistajien omia tai JEDECin määrittelemiä rajapintoja on useita, mutta ne voivat määritellä käytännössä koko muistipiirin toiminnallisuuden. Elektroniikka-alan ammattilehdessä Electronic Design [52] on julkaistu artikkeli NAND-muistien rajapinnoista muistivalmistaja Toshiban avustuksella. Artikkeli on hyvin lyhyt, mutta se kiteyttää NANDien rajapintatilanteen hyvin yhteen taulukkoon, joka on oleellisilta osin esitetty tässä taulukkona 4. Siitä voidaan lukea, että NAND-muistien rajapinnat voidaan lukea kolmeen kategoriaan: puhtaaseen NANDiin, valmistajien omiin rajapintoihin ja JEDEC-standardoituihin rajapintoihin. Puhdas NAND tarkoittaa pelkkää muistipiiriä, eli käytännössä rinnakkaista tiedonsiirtoa ja sitä, että järjestelmän pitää huolehtia virheenkorjauksesta ja muista perinteisistä muistiohjaimen toiminnoista. Toisin sanoen isäntäjärjestelmässä tulee olla muistiohjain. Aiemman luvun 3.3 muistivertailussa puhtaat NAND-muistit ovat sähköisesti ja käskykannaltaan samankaltaisia, mutta eivät identtisiä. JEDEC-standardoiduilla rajapinnoilla lähtökohta on täysin päinvastainen: kaikki muistiohjaimen toiminnot ovat rajapinnasta katsottuna muistipiirin puolella, joten isäntäjärjestelmän suunnittelussa voidaan keskittyä sen ydintoimintoon. JEDEC-standardeista *Sulautettu muistikortti (e·MMC, engl. Embedded MultiMediaCard)* on rinnakkaistyyppinen, kun taas *Yleinen Flash-muisti (UFS, engl. Universal Flash Storage)* on erittäin nopea sarjamuotoinen rajapinta. Puhtaan NANDin ja JEDEC-standardien väliin jää joukko muistivalmistajien omia standardeja. Nämä voivat sisältää mitä tahansa muistiohjaimen ominaisuuksista. Artikkelin mukaan erityisesti on tavallista, että ne sisältävät laitteistopohjaisen virheenkorjauksen, koska sen tekeminen ohjelmistolla on hidasta. Tällaisessa tapauksessa järjestelmän tehtäväksi jää kulutuksentasaus siihen liittyvine roskienkeräyksineen sekä viallisten lohkojen hallinta. Taulukossa 2 viitattiin *Avoimeen NAND Flash -rajapintaan (ONFI, engl. Open NAND Flash Interface).*

Kyseessä on alemman tason rajapinta, joka ei ota kantaa muistikontrolleriominaisuuksiin, vaan pääasiassa pyrkii yhdenmukaistamaan varsinaisten muistipiirien jalka-asettelun ja käytetyt signaalit, jotta esimerkiksi muistikorttivalmistajat voisivat helpommin vaihtaa NAND-piirin toiseen mahdollisimman pienellä vaivalla [53].

Tässä luvussa käsiteltiin yleisesti Flash-muistien olemassa olevia rajapintoja ajurikehityksen kannalta. Huomattiin, että sähköisesti rajapinnaltaan samankaltaiset NOR-muistit ovat näytejoukon perusteella usein rajapinnaltaan samankaltaisia. NANDeillä tilanne on kirjavampi, mutta selkeitä rajapintastandardeja on olemassa. Vaikka yhtä ainoaa Flash-muistien rajapintastandardia ei ole, alalla on otettu askelia rajapintojen yhtenäistämiseksi.

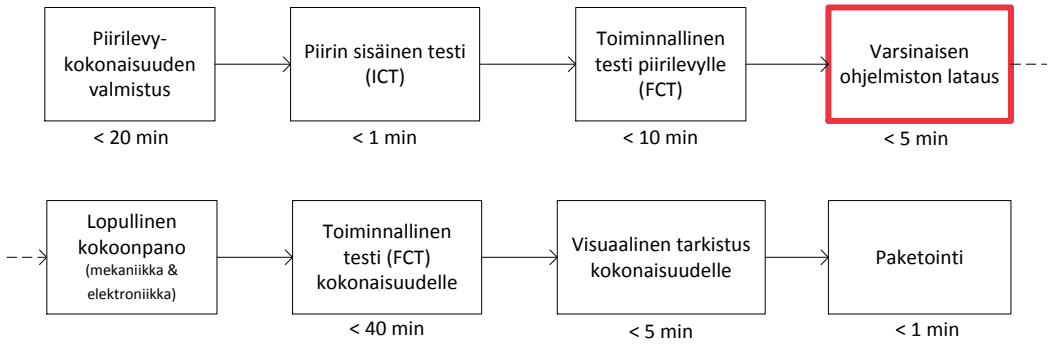
4 Taajuusmuuttajien massatuotanto

Ohjelmiston latausaseman suunnittelun motivaation ja tiettyjen ominaisuuksien ymmärtämiseksi on hyödyllistä tutustua lyhyesti taajuusmuuttajien valmistusympäristöön, jonne latausasema tullaan sijoittamaan. Tässä luvussa selitetään missä vaiheessa valmistusprosessia ohjelmisto syötetään taajuusmuuttajaan niillä taajuusmuuttajilla, joita asema tukee, ja millaisia henkilöitä ohjelmiston latauksen parissa työskentelee.

4.1 Tuotantolinja

Tuotantolinjalla tarkoitetaan tässä tekstissä tehdasta, jossa taajuusmuuttajat kootaan. Tässä vaiheessa elektroniikan komponentit on ladottu jo piirilevylle ja piirilevy komponentteineen voidaan käsittää yhtenä perusosana. Samaan tapaan lopullisen laitteen tehoelektroniikka, ohjelmisto ja mekaaninen kotelo saadaan yksinä osina.

Kokoonpano tuotantolinjalla on yksinkertaisimmillaan elektroniikan, ohjelmiston sekä mekaniikan yhdistämistä. Tämän lisäksi osien toimivuus testataan. Kuvassa 9 on eritelty työvaiheet yksityiskohtaisemmin. *Piirin sisäisessä testissä (ICT, engl. In-Circuit Test)* varmistetaan, että elektroniikka on valmistettu oikein. Oikeellisuus voidaan todentaa mittaamalla sähkönjohtavuus eri pisteiden välillä tai yksittäisten komponenttien parametreja. *Toiminnallisessa testissä (FCT, engl. Functional Test)* tarkistetaan laajemmin elektroniikan toimivuus. Testattavaan laitteeseen ladataan testausrutiineja sisältävä ohjelmisto ja laitteen eri toimintojen oikea toiminnallisuus varmistetaan vertaamalla tietyillä syötteillä saatuja lopputuloksia oletettuihin lopputuloksiin. [54, s. 377–378] Seuraavaksi laitteeseen ladataan sen lopullinen ohjelmisto, mikä on työssä suunniteltavan aseman tehtävä. Vaihtoehtoisesti lopullinen ohjelmisto voi sisältää myös testausrutiinit, jolloin se ladataan ennen toiminnallista testiä ja erillistä testiohjelmistoa ei tarvitse ladata lainkaan. Lopuksi ohjelmoitu ohjauskortti, tehoelektroniikka ja mekaniikka yhdistetään, tehdään lopulliset testit kokonaisuudelle ja paketoidaan valmis tuote.

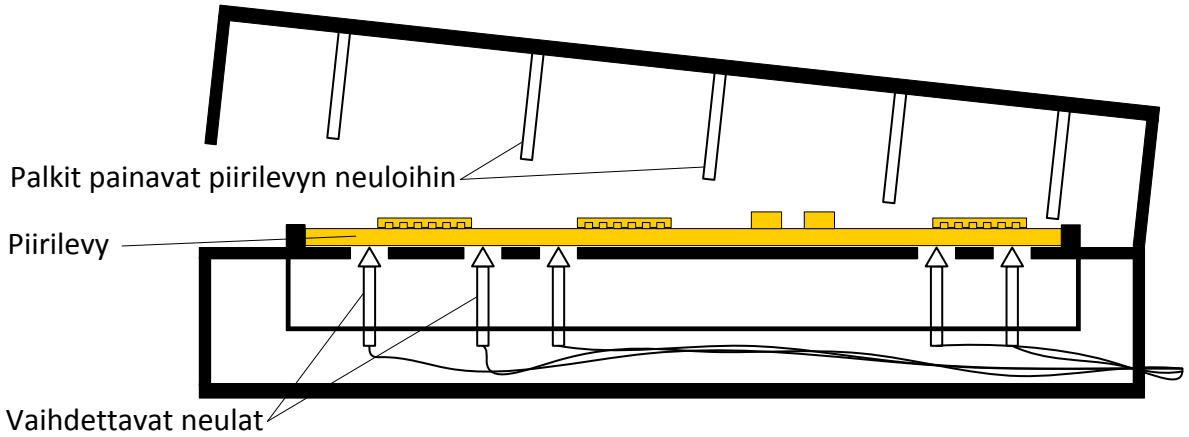


Kuva 9: Taajuusmuuttajan tuotantolinjasto vaiheittain ohjauskortin näkökulmasta.

Osa kuvassa 9 mainituista eri vaiheiden kestoista on pitkiä, jolloin ne määräävät yksittäisen taajuusmuuttajan valmistamiseen kuluvan ajan erityisesti sen jälkeen, kun tässä työssä suunniteltava latausasema valmistuu. ABB:lla on käynnissä itsenäisiä projekteja, joilla osaa näistä työvaiheista tehostetaan. Tuotantokapasiteettia voidaan myös

nostaa lisäämällä työntekijöitä ja tuotantolaitteita vaiheisiin, joita varten ei kehitetä uutta laitteistoa tai ohjelmistoa.

Neulapeti on tärkeä rakenne tuotantolinjalla. Niitä käytetään useissa vaiheissa testausprosessia, koska kortin kiinnittäminen niihin ja niistä pois on nopeaa, vaikka kytkettäviä liitoskohtia on kymmeniä. Neulapedin rakenne muistuttaa liitintä, joka kiinnittyy piirilevyn testipisteisiin mekaanisesti, kuten kuvassa 10 on havainnollistettu. Neulojen lisäksi oleellista on piirilevyn oikeaan sijaintiin ohjaavat ulokkeet sekä levyn neuloihin painava mekaniikka. Näitä tarvitaan tukevan liitännän varmistamiseksi. Neulat ovat jousitettuja tasaisen kiinnityksen aikaansaamiseksi ja ne tarttuvat piirilevyyn vasta kun neulapedin kansi suljetaan. Tällöin neulat eivät hankaloita testattavien tai ohjelmoitavien piirilevyjen asettamista petiin, kun kansi on auki.



Kuva 10: Neulapedin mekaaninen rakenne. Piirilevyn neuloihin painavan kannen sijaan piirilevy voidaan imeä neuloihin paineilmalla. [54, s. 377]

4.2 Työntekijäroolit ohjelmiston latauksessa

Ohjelmiston latausaseman ympärillä työskentelee kolme eri käyttäjäroolia: operaattori, huoltohenkilö ja insinööri. Näistä operaattori ja huoltohenkilö voivat olla tarvittaessa sama henkilö. Insinöörin tehtävät ovat luonteeltaan huomattavasti teknisempiä ja harvemmin suoritettavia, joten insinööri oletetaan olevan eri henkilö.

Operaattori on latausaseman peruskäyttäjä. Hänen tehtävänsä on asettaa ohjelmoitava elektroniikka laitteeseen, käynnistää lataus ja siirtää elektroniikka seuraavaan vaiheeseen tuotantolinjalla. Tehtävä ei vaadi juurikaan koulutusta. Operaattori viettää latausaseman kanssa käytännössä koko työpäivänsä, joten suunnitteluvaiheessa huomioidaan työergonomia ja aseman käytettävyys hänen näkökulmastaan.

Huoltohenkilö toimii latausaseman parissa säännöllisesti, mutta harvemmin kuin operaattori. Hänen tehtävänsä on muun muassa vaihtaa aseman neulapeti, kun halutaan ohjelmoida toisentyyppisiä piirilevyjä, ja säätää laitteen ohjelmiston asetuksia tarvittaessa. Hän ratkoo myös mahdollisia yksinkertaisia ongelmatilanteita. Tehtävä on haastavampi kuin operaattorin työ, mutta ei niin haastava, etteikö operaattorina toimiva henkilö voi hoitaa myös huoltohenkilön roolia, jos hänet koulutetaan tehtävään.

Insinööri jatkokehittää latausasemaa edelleen, kun asemaan tarvitaan uusia ominaisuuksia. Tämä tarkoittaa esimerkiksi neulapedin neulojen uudelleensijoittamista, kun uudentyyppisiä piirilevyjä halutaan ohjelmoida. Muihin oletettaviin jatkokehityskohtiin palataan myöhemmin luvussa 6.6. Työ on teknisesti haastavampaa kuin muut työntekijäroolit, mutta insinööriä tarvitaan harvemmin.

4.3 Nykytilanne ja tavoitteet

Osaa suunniteltavan latausaseman tukemista ohjauskorteista valmistetaan jo työn kirjoitushetkellä. Ohjelmiston syöttämiseen käytetään testerilaitetta, joka syöttää ohjelmiston yhteen taajuusmuuttajan ohjauskorttiin kerrallaan. Yhden operaattorin vastuulla on kaksi tällaista konetta. Suunniteltava latausasema on fyysisesti lähes samankokoinen, mutta tukee yhden ohjelmoitavan kortin sijasta kuutta korttia (lisätietoja luvussa 6). Jos yksi operaattori on pystynyt käsittelemään kahta latausasemaa aiemmin, pystyy hän varmasti tähän myös jatkossa. Tavoite on siis kasvattaa työskentelyn tehokkuus kuusinkertaiseksi.

Työtehon kasvatus on tarpeen, sillä tulevaisuudessa tarvitsee ohjelmoida moninkertainen määrä ohjauskortteja. Uusien taajuusmuuttajien valmistusprosessissa on vaiheita, joita tehostetaan ennen kuin kyseinen taajuusmuuttajamallisto saavuttaa siltä odotettavat myyntivolyymit. Yksi tehostettavista vaiheista on ohjelmiston lataus, joka on tämän työn aihe.

5 Vaatimukset ohjelmiston latausasemalle

Työn toimeksiantaja asetti seuraavassa luvussa suunniteltavalle ohjelmiston latausasemalle useita vaatimuksia. Osaa ominaisuuksista toivottiin jo heti työn alkaessa, mutta osa ilmeni vasta tarkemmin tarvetta selvitettäessä. Seuraavat vaatimukset esitettiin alun perin yksinkertaisemmin, mikä voi aiheuttaa eriäviä tulkintoja. Listassa vaatimuksia on tarkennettu siten, että ne vastaavat toimeksiantajan tarkoittamaa yksiselitteisesti.

1. *Kortit helposti liitettäviä*: Kokenut käyttäjä kykenee asettamaan ohjelmoitavat kortit asemaan ja ottamaan ne sieltä pois korkeintaan kymmenessä sekunnissa ilman henkilö- tai materiaalivahinkoja. Henkilövahinko tarkoittaa mitä tahansa terveyden kannalta negatiivista vaikutusta, joka aiheutuu korttien liittämisestä asemaan tai irrottamisesta siitä. Esimerkiksi sormien kipeytyminen on henkilövahinko. Materiaalivahinko on latausaseman tai liitettävän kortin rikkoutuminen, mukaan lukien minkä tahansa liittimen vahingoittuminen, vaikkei rikkoutuminen vaikuttaisi liittimen toiminnallisuuteen.

2. *Helposti opittava*: Toimeksiantajan työntekijä voidaan kouluttaa latausaseman operaattoriksi helposti. Helpolla tarkoitetaan tässä, että tulevan operaattorin esimies tai kokenut työtoveri voi opettaa aseman käytön alle tunnissa.

3. *Revisio määritettävissä*: Latausaseman kaikkien suunniteltavien ohjelmistojen versionumero on saatavilla huoltomiehelle sekä insinöörille. Saatavuudella tarkoitetaan, että latausaseman pitää pystyä ilmoittamaan eri versionumerot esimerkiksi asetusnäytössä.

4. *Tuki useille ohjauskorteille*: Asemalla voi ohjelmoida eri ohjauskorttityyppejä sekä niiden eri sähköisiä revisioita, ja ohjelmoitavan ohjauskorttityypin vaihto saa kestää korkeintaan 15 minuuttia. Alustavasti eri ohjauskorttityyppejä on kaksi, mutta niitä tulee myöhemmin lisää määrittämätön lukumäärä erilaisia. Kerrallaan ohjelmoidaan vain yhden tyyppisiä kortteja.

5. *Tuki useille korttien revisioille*: Asema voi syöttää eri ohjelmistoversion ohjauskorttiin sen revision perusteella. Eri revision ohjauskortteja tulee voida ohjelmoida samanaikaisesti, sikäli kuin niiden latausaseman kannalta oleelliset liitäntärajapinnat[16] ovat yhtenevät.

6. *Toiminnan jäljitettävyys*: Ohjelmoitujen korttien sarjanumerot, operaattorin tunnus, ajankohta, ohjelmoinnin kesto sekä onnistuminen kirjataan lokiin. Lokitiedoston muotoilu noudattaa toimeksiantajan määritystä, jota ei käsitellä tässä työssä.

Seuraavia vaatimuksia ei voida todeta tämän työn puitteissa onnistuneiksi tai epäonnistuneiksi. Ne annettiin ohjaamaan suunnittelua, joten ne listataan tässä täydellisyyden vuoksi.

[16]Liitäntärajapinnalla tarkoitetaan vaihtoehtoisesti ohjelmointiin käytettyä sähkömekaanista liitintä tai neulapedin tapauksessa sitä testipisteiden osajoukkoa, jota latausasema käyttää.

1. *Luotettava*: Latausasema syöttää ohjelmiston ohjauskortille yhdellä yrityksellä yhtä suurella onnistumistodennäköisyydellä kuin mitä sitä edeltävä latausasema pystyi. Tämän testaamiseksi saatavilla ei ole riittävää näytejoukkoa ohjauskortteja. Tämä voidaan todentaa vasta, kun latausasema on tuotantokäytössä.

2. *Kestävä rakenne*: Aseman mekaaninen rakenne kestää jatkuvaa käyttöä vähintään toimeksiantajan määrittämän tavallisen tuotantolaitteen käyttöiän. Käyttöikä on annettu yleisesti kaikille toimeksiantajan tuotantolaitteille tiettynä käyttökertojen lukumääränä ja käyttövuosina. Tämän onnistuminen voidaan todentaa vasta määrätyn ajan jälkeen.

3. *Helposti huollettava*: Latausaseman kuluviin osiin päästään helposti käsiksi. Tässä tapauksessa helpolla tarkoitetaan, että aseman huoltamiseksi sitä ei tarvitse purkaa osiin, vaan yksittäisten ruuvien tai kiinnikkeiden irrottaminen riittää. Tarkka ruuvien tai kiinnikkeiden lukumäärä on mekaniikan valmistavan alihankkijan päätettävissä. Tämä kohta voidaan todentaa vasta, kun alihankkija esittää piirustukset mekaniikasta.

Edellä mainittujen vaatimusten ja ohjeiden lisäksi annettiin laadullisia suuntaviivoja, joita ei voida todeta onnistuneiksi tai epäonnistuneiksi edes myöhemmin. Seuraavassa listatut suuntaviivat auttavat ymmärtämään myöhemmin tekstissä perusteltuja valintoja.

1. *Ergonominen*: Operaattorin terveys ei heikkene pitkällä aikavälillä huonon työasennon vuoksi.

2. *Rajoitettu koko, erityisesti leveys*: Asema saa käyttää tehtaan lattiapinta-alaa muihin tuotantolaitteisiin verrattavissa olevan määrän. Tämä on listattu ei-todettavaan kategoriaan, koska kyseessä on enemmän toive kuin vaatimus: tehtaassa riittää tilaa, mutta erityisesti jos latausasema on leveä, muu linjasto joudutaan suunnittelemaan aseman ympärille, mikä ei ole toivottavaa.

3. *Laskennallisesti nopea*: Ohjelmointiaika jaettuna ohjelmoitavien ohjauskorttien lukumäärällä tulee olla pieni. Latausaseman nopeus on yhtäältä sen tärkein ominaisuus, mutta toisaalta sen mittaaminen ei ole suoraan mahdollista. Laskennallista nopeutta saadaan kasvatettua lisäämällä paikkoja ohjelmoitaville ohjauskorteille, mutta toisaalta tästä aiheutuu ongelmia ergonomian ja fyysisten kokorajoitteiden kannalta. Kyseessä on laadullinen monimuuttujafunktio.

Latausasemalle annettiin useita vaatimuksia, ohjeita ja suuntaviivoja, joista vain osaa voidaan mitata suoraan. Seuraavassa sekä luvussa 7 palataan tarkemmin tässä luvussa mainittuihin aseman ominaisuuksiin.

6 Latausaseman suunnittelu

Tässä luvussa esitellään työn käytännön osuutena suunniteltava ohjelmiston latausasema sekä perustellaan tehtyjä valintoja ja käydään läpi vaihtoehtoisia toteutustapoja. Luku alkaa aseman konseptin esittelyllä ja jatkuu eri osa-alueiden käsittelyllä. Osa-alueet noudattavat projektin työnjakoa: tässä diplomityössä suunnitellaan elektroniikka, mikrokontrolleriohjelmisto sekä tietokonekäyttöliittymä. Mekaniikka alihankitaan ja myöhemmin perustelluista valinnoista johtuen ohjelmiston lataukseen käytetään olemassa olevaa latausrutiinia. Lopuksi eri osa-alueiden ylläpidettävyyttä mietitään siitä näkökulmasta, että kuinka helppoa toimeksiantajan henkilöstön on ylläpitää diplomityönä suunniteltua latausasemaa tulevaisuudessa.

Yksi ylläpidettävyyden määritelmä on, että järjestelmää voidaan muuttaa joko virheiden korjaamiseksi tai uusien vaatimusten huomioimiseksi [55]. Tämä näkemys ei ota kantaa siihen, kuinka työlästä muutoksien tekeminen on. Eick [56] laajentaa ylläpidettävyyden määritelmää julkaisussaan huomioimalla siitä aiheutuvat kustannukset. Hän on analysoinut laajaa puhelinkeskuksen sulautetun järjestelmän ohjelmistoa ja esittää, että suurin osa ylläpidosta koostuu muuttuvaan ympäristöön mukautumisesta ja uusien ominaisuuksien toteuttamisesta. Muutoskustannukset tarkoittavat ohjelmiston yhteydessä käytännössä insinöörin palkkaa, mutta kustannukset nousevat käytettyjen työtuntien mukaan. Latausaseman yhteydessä on huomattava vielä, että sen valmistuttua vastuu siitä jää henkilölle tai tiimille, jolla on oletettavasti suuri työtaakka jo valmiiksi. Kaikki muutostyöt ovat pois muusta työstä. Tässä tekstissä ylläpidettävyydellä tarkoitetaan jälkimmäistä näkemystä, joka huomioi muutosmahdollisuuden lisäksi sen aiheuttamat kustannukset.

6.1 Konsepti

Ohjelmisto voidaan syöttää aseman kannalta oleellisten taajuusmuuttajien ohjauskorttien muistipiireihin seuraavilla eri tavoilla.

1. Testipisteiden kautta muistipiirille.
2. Testipisteiden kautta kommunikoidaan pääsuorittimen kanssa ja se kirjoittaa tiedon muistipiirille.
3. Perinteisen sähkömekaanisen liittimen kautta kommunikoidaan pääsuorittimen kanssa ja se kirjoittaa tiedon muistipiirille.

Latausaseman ei tarvitse kirjoittaa pääsuorittimen sisäiseen muistiin *alkulatausohjelmaa (engl. bootloader)*, joka tarvitaan pääsuorittimen kanssa kommunikointiin. Tämä on ladattu jo aiemmin tuotantolinjalla toiminnallisessa testissä.

Ensimmäinen vaihtoehto on ehdottomasti nopein. Jos oletetaan taulukossa 2 esitettyjen sarjamuotoisten NORien edustavan ominaisuuksiltaan toimeksiantajan käyttämiä NOR-muisteja, muistin tyhjentäminen, ohjelmiston kirjoittaminen ja sen oikeellisuuden

varmistaminen voidaan suorittaa alle kahdessa minuutissa[17]. Ongelmia ovat ratkaisun toteuttamisen vaikeus ja työläs ylläpidettävyys. Työn tekoaikaan muistipiirille ladattavasta ohjelmistosta ja oheistiedostoista ei ollut saatavana kokonaista *levykuvaa (engl. image)*, joten levykuvan kokoaminen ja muistin alustaminen oikeanlaiseksi on monimutkaista. Tästä seuraa, että kaikkia ohjelmistovirheitä ei välttämättä löydetä alustavissa testeissä ja niiden korjaaminen jää ABB:n henkilökunnan tehtäväksi. Työn ollessa jo pitkällä, nousi esiin, että levykuva voitaisiin lukea jo-ohjelmoidun taajuusmuuttajan muistipiiristä. Aikataulun vuoksi tätä vaihtoehtoa ei tutkittu pidemmälle, mutta mahdollinen ongelma on, että koska taajuusmuuttajan muistin sisältö on salattu, aiempien testivaiheiden testitulokset voivat hävitä, kun niiden päälle kirjoitetaan uusi levykuva.

Toisessa ja kolmannessa vaihtoehdossa voidaan hyödyntää ABB:n ohjelmistotiimin ylläpitämää latausohjelmaa. Latausohjelma on komentoriviohjelma, joka syöttää halutun revision ohjelmistosta yhteen taajuusmuuttajaan. Ohjelmisto toimii kommunikoimalla suorittimen kanssa ja jättämällä muistipiirin käsittelyn suorittimen vastuulle. Aikaa kuluu enemmän kuin ensimmäisessä vaihtoehdossa. Tämä on kuitenkin selkeästi helpompi vaihtoehto jatkossa, koska se ei aiheuta uusia ohjelmiston ylläpitotöitä kenellekään, vaan ohjelmistotiimi päivittää omaa ohjelmaansa kuten aiemminkin välittämättä latausasemasta. Toimeksiantajan komentoriviohjelmisto huolehtii myös kirjoitetun tiedon oikeellisuuden tarkistamisesta.

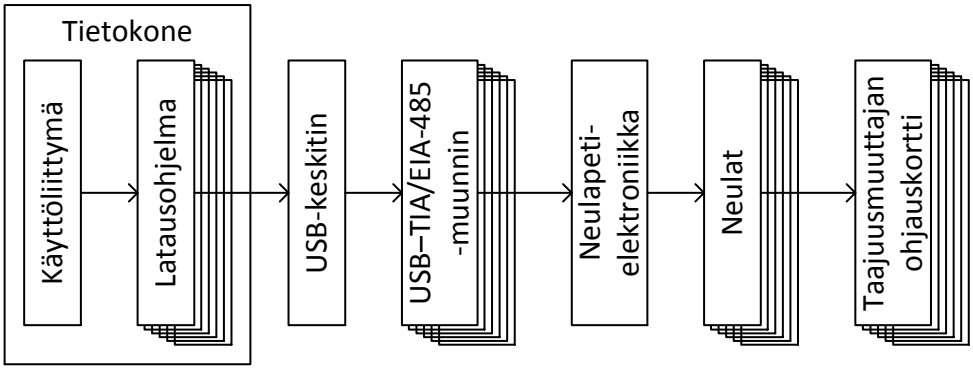
Sähkömekaanisten liitinten käyttäminen keventää aseman mekaniikkaa neulapetiin verrattuna, mutta niiden kiinnittäminen ja irrottaminen on hidasta, ja huolimaton operaattori saattaa rikkoa liittimen. Neulapedin suunnittelu vaatii siihen erikoistuneen mekaniikkasuunnittelijan, mutta rakenteen käyttäminen on nopeaa ja operaattori ei tarvitse enempää huolellisuutta kuin tavallisesti piirilevyä käsiteltäessä. Molemmissa vaihtoehdoissa liittimet kuluvat ja ne tulee vaihtaa tietyin määräajoin. Tuotantolinjalla on kuitenkin muitakin neulapetilaitteita ja koulutettu henkilöstö, jonka normaaliin työnkuvaan kuuluu neulojen vaihtaminen, joten aseman huoltaminen vaatii vain vähän lisätyötä tuotantolaitteiden ylläpitohenkilöstölle.

Yllä olevista vaihtoehdoista valittiin vaihtoehto numero kaksi. Ratkaisulla saadaan helposti ylläpidettävä laite hieman nopeuden kustannuksella.

Latausprosessi on yksittäisen latausohjelman ja ohjauskortin välinen keskustelu. Yksittäisellä ohjauskortilla voi kestää tiedon prosessoinnissa aavistuksen eri aika kuin toisilla tai se voi joutua pyytämään tietopakettia uudestaan, jos lähetyksessä tapahtui virhe. Ohjelmisto lähtee tietokoneelta ohjauskorteille *Yleissarjaväylää (USB, engl. Universal Serial Bus)* pitkin kuvan 11 mukaisesti. Koska samaa signaalia ei voida syöttää kaikille korteille, tietokoneelta lähetetään kuusi korttikohtaista signaalia yhdellä kaapelilla USB-keskittimelle, jossa se jaetaan kuudelle USB-kaapelille. Sen jälkeen jokainen USB-signaali muunnetaan taajuusmuuttajan ymmärtämään muotoon, joka on tässä tapauksessa TIA/EIA-485[18]. Muunnos tehdään ABB:n kehittämällä

[17]Kappaleessa 3.3 esiteltyjen sarjamuotoisten NORien tyypillisiä arvoja käyttäen, koko muistin ohjelmointi jakautuu seuraavasti: koko piirin tyhjennys 30–68 s, koko piirin kirjoitus 13–40 s, kirjoituksen oikeellisuuden varmistus alle 1 s. Yhteensä ohjelmiston lataus kestää 43–109 s.

[18]TIA/EIA-485 tunnettiin aiemmin nimellä RS-485. TIA on *Tietoliikenneteollisuuden yhdistys (engl. Telecommunications Industry Association)*, *EIA on Elektroniikkateollisuuden liittouma (engl. Electronic Industries Alliance)* ja RS on *Suositeltu standardi (engl. Recommended Standard)*.



Kuva 11: Ladattavan ohjelman polku tietokoneelta taajuusmuuttajaan. Kuvassa on esitetty rinnakkaiset osat pinoina.

USB–TIA/EIA-485-muuntimella sen helpon saatavuuden vuoksi, mutta se voitaisiin tehdä tarvittaessa myös muulla kyseistä muunnosta tukevalla laitteella. Seuraavaksi datasignaalin rinnalle tuodaan tarvittava käyttöjännite ennen neulapetiä. Tämä tapahtuu myöhemmin tässä luvussa suunniteltavalla neulapetielektroniikalla. Lopuksi sekä käyttöjännite että datasignaali viedään taajuusmuuttajaan neulapedin neulojen avulla.

Latausasema valittiin tukemaan yhtäaikaisesti kuuden ohjauskortin ohjelmointia. Päätökseen vaikutti huomattavasti ABB:n näkemys siitä, että 4–5-paikkainen asema olisi riittävä aiemmalle tuoteperheelle, jonka tuotantomäärät tosin ovat matalammat kuin mitä uudelta tuoteperheeltä toivotaan. Toisaalta kokoonpanolinjastolla yhdelle laitteelle varattu lattiapinta-ala on rajallinen, joten kovin suurta laitetta ei ole käytännöllistä rakentaa. Kuusi ohjauskorttia mahtuu verrattain kompaktiin tilaan[19] ja tarjoaa riittävän laskennallisen ohjelmointinopeuden. Koska asema ei ole normaalia tuotantolinjan laitetta suurempi, niitä voidaan tarvittaessa rakentaa muutama kappale, jolloin suurikin vuosituotanto saadaan katettua.

Latausasemassa on myös joukko lisäominaisuuksia, joiden tarkoitus on helpottaa laitteen käyttöä ja vähentää virhetilanteita, jotka voivat johtaa taajuusmuuttajien tai käyttäjien vahingoittumiseen. Vaikka ohjelmoitavat kortit toimivat 5 V jännitteellä, jota pidetään turvallisena eikä se täten vaadi erityisiä turvallisuushuomioita, varmuudeksi asema monitoroi kanttansa. Jos se avataan, sähköt katkaistaan osista, joihin käyttäjä voi koskea. Näin käyttäjä ei pysty satuttamaan itseään vahingossa. Lisäksi aseman kansi on latauksen aikana lukittu, jotta materiaalivahingoilta vältytään. Latausprosessissa on hetki, jolloin jos latausprosessi keskeytyy, ohjauskortin suorittimesta tulee käyttökelvoton. Laitteessa on myös *valodiodeja (LED, engl. Light-Emitting Diode)*, jotka näyttävät käyttäjälle yksittäisen kortin ohjelmiston latauksen tilan.

Viimeisenä osana laitetta on taajuusmuuttajille riittävän virran tarjoaminen. Yksi ohjauskortti vie korkeintaan 1 A virtaa 5 V jännitteellä, kun siihen syötetään ohjelmistoa. Tarvittava virta saadaan laboratorioteholähteestä. Lisäksi tarvitaan 24 V jännite solenoidia varten laboratorioteholähteestä. Vaihtoehtoisesti riittävästi virtaa saadaan USB-keskittimen kautta, kun yksi USB–TIA/EIA-485-muunnin kytketään kahteen USB

[19]Erilaisista mekaanisista ratkaisuista, jotka mahtuvat pieneen tilaan, mutta joihin mahtuu monta ohjelmoitavaa korttia, on selitetty mekaniikkaa käsittelevässä luvussa 6.2.

2.0 -porttiin. USB 2.0 spesifikaation mukaan yksi portti tarjoaa vähintään 500 mA virtaa 4.75–5.25 V jännitteellä [57]. Standardin sanamuodosta huolimatta, käytännössä USB 2.0:n virraksi oletetaan 500 mA eikä sitä enempää.

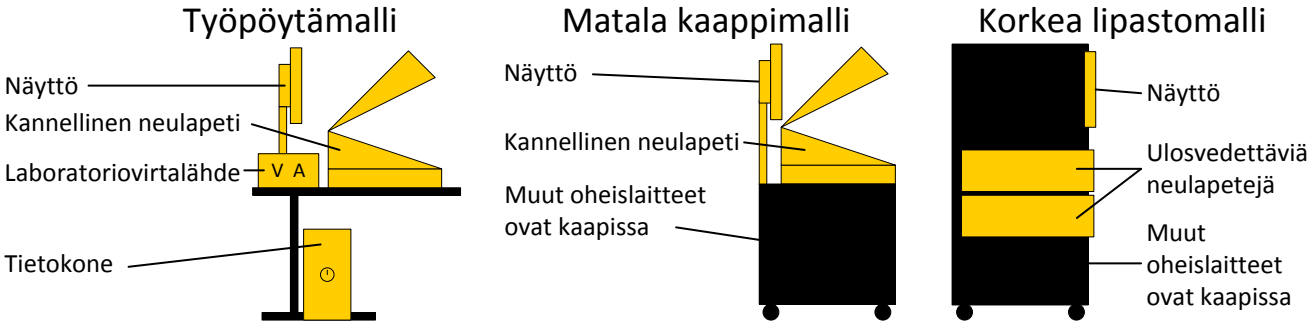
Lyhyesti latausaseman konsepti on, että olemassa olevaa latausohjelmaa ajetaan tietokoneella rinnakkain, signaali muunnetaan taajuusmuuttajalle sopivaan muotoon ja yhdessä riittävän virran kanssa viedään ohjauskortille. Perusidea on yksinkertainen ja toimeksiantajalle kevyt ylläpitää, mutta jotta laitteesta saadaan oikeasti hyödyllinen ja pitkäikäinen, se vaatii lisäominaisuuksia ja suunnittelua. Näitä näkökulmia on avattu seuraavissa luvuissa.

6.2 Mekaniikka

Mekaniikan yksityiskohtainen suunnittelu ja toteutus alihankittiin, joten se ei kuulu työn piiriin samassa laajuudessa kuin elektroniikka ja ohjelmistot, mutta mekaniikan yleissuunnitelma täytyy hahmotella, jotta alihankkija voi tehdä osuutensa. Lisäksi tässä luvussa esitettävät kuvat auttavat havainnollistamaan, minkälaiseen asemaan seuraavan luvun elektroniikka sijoitetaan.

Luvussa 5 mainittiin latausasemaan liittyviä vaatimuksia ja ohjeita, joista osa liittyy mekaniikkaan. Mekaniikka voidaan suunnitella siten, että korttien nopean liitettävyyden ja eri ohjauskorttityyppien vaihdettavuuden onnistuminen voidaan mitata, mutta sen rakennetta, huollettavuutta, ergonomiaa ja fyysisen koon optimaalisuutta ei voida mitata riittävällä tarkkuudella, että niiden onnistumisesta voitaisiin sanoa mitään. Voidaan kuitenkin esittää, että miten ne otettiin huomioon suunnittelussa.

Latausaseman ergonomia ja optimaalinen fyysinen koko ovat tärkeitä operaattorille ja tuotantolinjalle. Ergonomia on tärkeää sen vuoksi, että operaattori työskentelee päivittäin latausaseman kanssa ja ylimääräinen kurottelu tai epäergonominen työasento ilmenee terveyshaittoina myöhemmin. Aseman sivuttaistilaan huomion kiinnittäminen on syvyyttä tai korkeutta tärkeämpää sen vuoksi, että tuotantolinjalla eri työvaiheet on sijoitettu vierekkäin, jolloin seuraava linjaston laite on niin lähellä viereistä kuin mahdollista ja minimoimalla leveys, samaan tilaan mahtuu enemmän tuotantolaitteistoa.



Kuva 12: Latausaseman mekaniikan konseptityypit, joihin kaikki harkitut mekaniikkarakenteet perustuivat.

Mekaniikkakonseptiin ideoita saatiin tuotanto- ja testilaitteiston kanssa tekemisissä olevilta ABB:n työntekijöiltä. Varteenotettavia vaihtoehtoisia konseptityyppejä oli kolme,

minkä lisäksi harkittiin myös näistä muunneltuja versioita. Kuvassa 12 on esitelty ideoiden päätyypit. Ensimmäinen on työpöydälle asetettava malli, jossa neulapeti ja elektroniikka ovat sen sisällä, ja laboratorioteholähteet sekä tietokone ovat erillään. Vaihtoehto on helppo toteuttaa ja kaikkein halvin, mutta sitä on hankala siirtää, koska osat pitää siirtää yksitellen ja se täytyy rakentaa aina uudestaan. Toisessa vaihtoehdossa kaikki tarvittava on matalan pyörillä olevan kaapin sisällä ja neulapeti sekä tietokoneen näyttö ovat kaapin päällä. Tätä mallia on helppo siirtää ja se on kompakti. Jottei operaattorin tarvitse kurotella turhaan, ohjelmoitavat piirilevyt täytyy asetella kahteen riviin, jonka vuoksi kaksi ensimmäistä vaihtoehtoa ovat hieman pienintä mahdollista leveämpiä. Kolmas vaihtoehto on korkea räkkihyllyyn rakennettava lipastomalli, joka sisältää kaiken tarvittavan. Räkkihyllyjä valmistetaan standardilevyisinä, minkä vuoksi se on oletuksena vaihtoehdoista kapein. Ohjelmoitavat kortit asetetaan ulos vedettäviin laatikoihin, mutta kapeudesta johtuen yksi laatikko ei riitä. Toisaalta ergonomian kannalta kaksi laatikkoa on ehdoton maksimi. Tämän vaihtoehdon suurimmaksi ongelmaksi ilmeni sen huollettavuus: ulosvedettävät laatikot vaativat verrattain monimutkaisia kuluvia mekaanisia rakenteita. Parhaaksi vaihtoehdoksi osoittautui vaihtoehto numero kaksi, eli kompaktin kaapin ja neulapedin yhdistelmä.

Erilaisia ohjauskortteja ohjelmoidaan eri neulapedeillä, jotka ovat vaihdettavissa adaptereissa. Niillä tarkoitetaan kuvassa 12 näkyvää kannellista neulapetiä. Kaikki adapterit ovat samankokoisia, mutta niiden sisältö vaihtuu. Ne kiinnitetään muuhun järjestelmään yhteisellä liitinrajapinnalla, jonka toisella puolella on matalan kaapin sisällä olevan elektroniikan johdot ja adapterin puolella jatko näille. Siten adapterin vaihto on nopeaa. Kun ohjauskorteista tehdään uusia revisioita tai latausasemaa halutaan laajentaa tukemaan uusia ohjauskortteja, ainoastaan uusi adapteri pitää valmistuttaa.

Kortit voidaan sijoittaa joko poikittais- tai pitkittäisasentoon yhteen tai useampaan riviin aseman sisälle. Kuitenkin jos piirilevyjä on yli kahdessa rivissä, operaattori joutuu kurkottamaan, mikä on ergonomian kannalta huono asia. Vierekkäin kortteja ei ole mielekästä laittaa yli kolmea, jottei asemasta tule liian leveää. Niiden asettaminen riviin pitkittäin, eli piirilevyjen pidemmät sivut vierekkäin, johtaa poikittaista asettelua kapeampaan lopputulokseen.

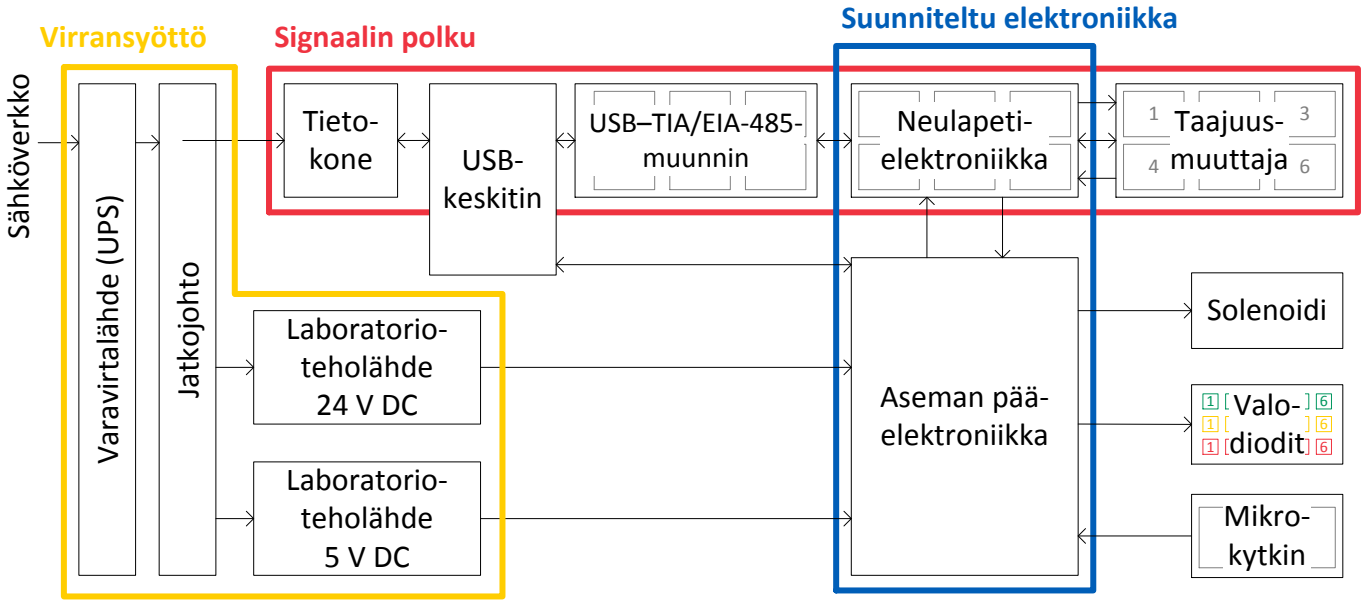
Vaikka lopullinen mekaniikkasuunnittelu ja sen toteutus jätettiin erillisen toimijan tehtäväksi, se suunniteltiin konseptitasolla sillä tarkkuudella, että tässä työssä suunniteltava järjestelmä sekä mekaniikka sopivat yhteen. Luvun avulla seuraavat kappaleet ja niissä tehdyt valinnat on myös helpompi hahmottaa, kun lukijalla on käsitys laitteen mekaanisesta rakenteesta.

6.3 Elektroniikka

Elektroniikan suunnittelu oli työn käytännön osuuden yksi keskeisimmistä osista. Tässä luvussa kuvataan elektroniikka lohko-, *kytkentä- ja sijoittelukaaviotasolla (engl. schematic and layout)*. Jälkimmäiset ovat kokonaisuudessaan liitteinä A ja B. Wilson [54] on listannut alan ammattilaisten kokemuksista nousseita piirisuunnittelun suositeltavia käytäntöjä kirjaksi. Latausaseman kannalta näistä tärkeimmät käsittelevät seuraavia asioita:

- Asettelukaavion *suunnittelusäännöt (engl. design rules)* johtimille ja reitittämiselle
- Häiriöiden vähentäminen *maatasolla (engl. ground plane)* ja *erotuskondensaattoreilla (engl. decoupling capacitor)*
- Komponenttien asettelu ja valinta
- Piirilevyn panelointi ja silkkipainetut merkinnät
- *Vahtikoiran (engl. watch dog)* käyttäminen mikrokontrolleriohjelmassa
- Operaattorin sähköturvallisuus
- *Hankinnan (engl. sourcing)* ja asentajien työmäärän huomioiminen

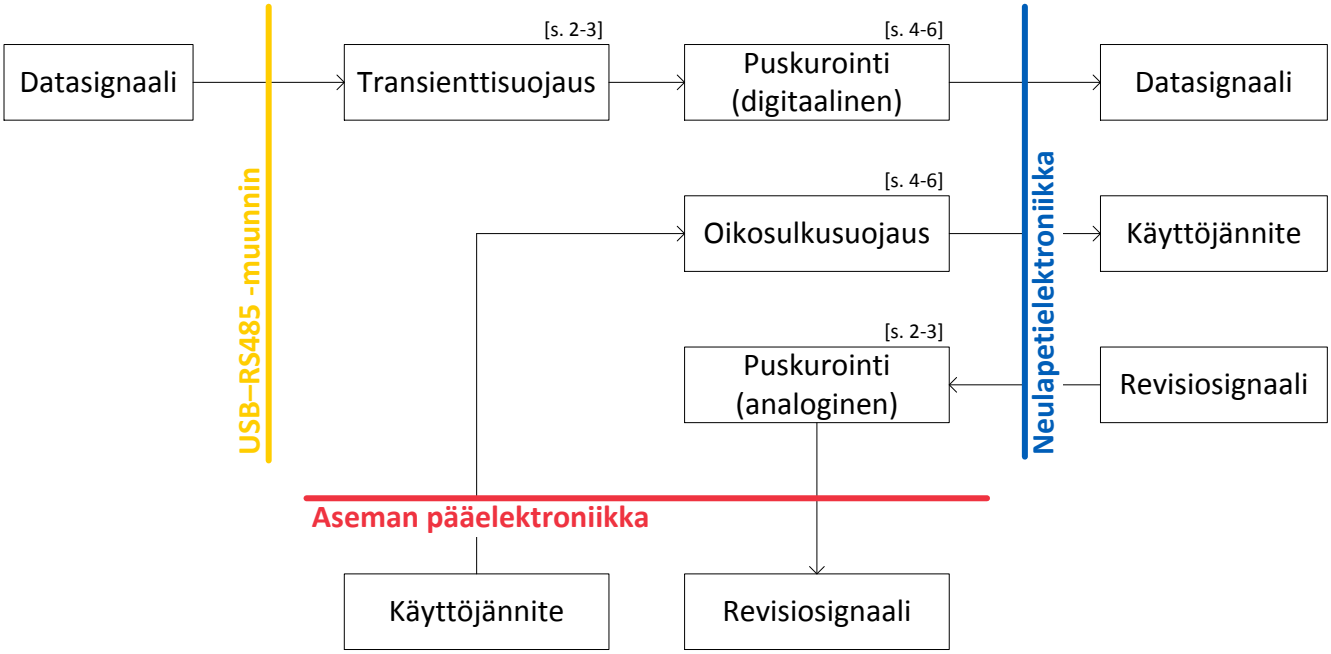
Latausaseman elektroniikkasuunnittelussa sovellettiin näitä käytäntöjä, joita avataan jäljempänä tässä luvussa. Lisäksi jokainen suunnitteluvaihe katselmoitiin toimeksiantajan käytänteiden mukaisesti.



Kuva 13: Koko latausaseman sähköjärjestelmä lohkokaaviona, josta näkee, mikä osa järjestelmästä suunniteltiin (”Suunniteltu elektroniikka” -lohko) ja mitkä hankittiin valmiina.

Aseman elektroniikka sisältää kaksi tässä työssä suunniteltavaa osaa. Lohkokaaviossa kuvassa 13 on vasemmalla sähkön syöttö. Pääasiallinen sähkö saadaan sähköverkosta, jonka jännitteellä ei ole merkitystä, kunhan se huomioidaan muissa latausasemaan asennettavissa osissa, eli tietokoneessa, USB-keskittimessä sekä laboratorioteholähteissä. Lyhyiden sähkökatkojen varalta käytetään *varavirtalähdettä (UPS, engl. Uninterruptable*

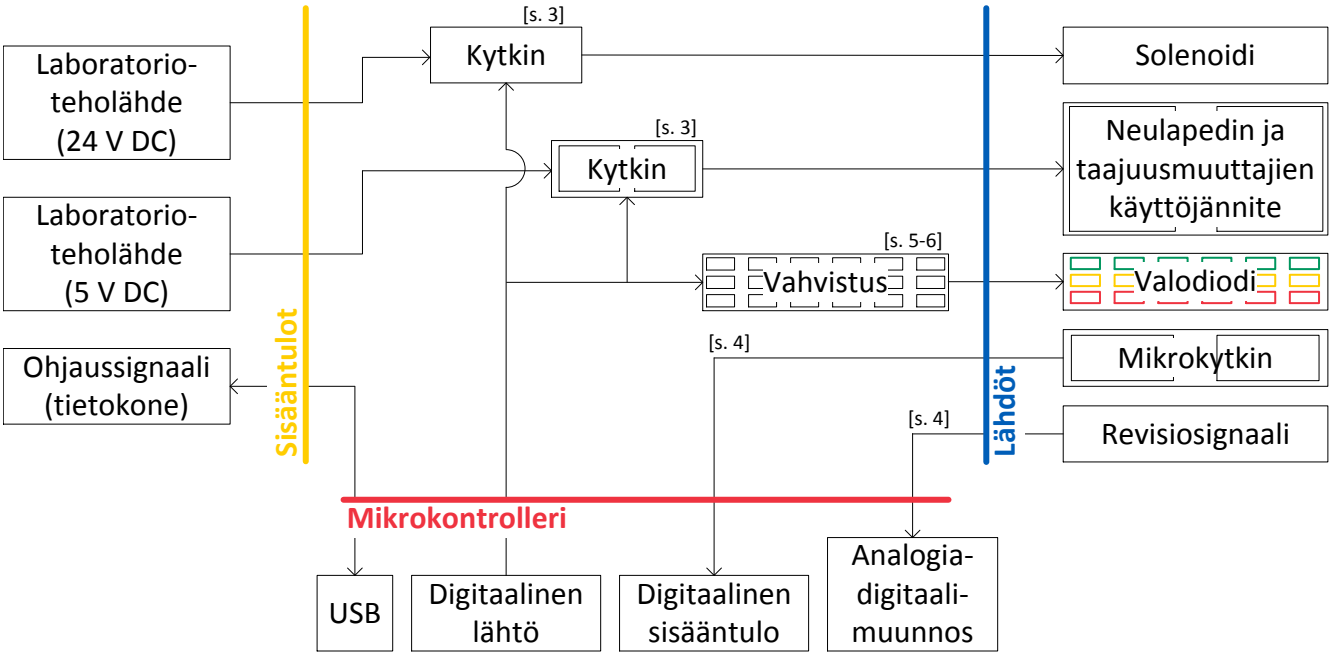
Power Supply), sillä ohjelmiston latauksessa on kriittinen vaihe, jolloin ohjauskortin prosessorista voi tulla käyttökelvoton sähköjen katketessa. Suunniteltava elektroniikka käyttää 5 V ja 24 V tasajännitettä, jotka saadaan verkkovirtaan kytkettävistä laboratorioteholähteistä. Lohkokaavion yläosassa on korostettu jo luvussa 6.1 kuvattu ohjelmiston polku tietokoneelta ohjelmoitaville taajuusmuuttajan ohjauskorteille. Suunniteltava elektroniikka on merkitty kuvaan ja sen vieressä on erillisiä sähköisiä ja sähkömekaanisia osia. Elektroniikka jaettiin kahdelle piirilevylle, koska ne oli tarkoitus sijoittaa eri paikkoihin aseman sisällä, mutta loppuvaiheessa mekaniikka muuttui alihankkijan ehdotuksesta, jonka seurauksena piirilevyt päädyttiin sijoittamaan vierekkäin ja kahtiajaosta ei hyödytty.



Kuva 14: Neulapetielektroniikan lohkokaavio. Lohkojen yhteydessä mainitut sivunumerot viittaavat kytkentäkaavion sivuihin liitteessä A.1.

Ensimmäinen suunniteltavista osista on neulapetielektroniikka, jonka päätehtävä on yhdistää alunperin tietokoneelta tuleva datasignaali sekä laboratorioteholähteiltä tuleva käyttöjännite ohjelmoitaville taajuusmuuttajille ohjelmiston kirjoituksen ajaksi. Käyttöjännitteen hallinta on jäljempänä selitettävässä toisessa suunniteltavassa osassa, joten neulapetielektroniikan tulee päästää datasignaali sekä käyttöjännite taajuusmuuttajalle aina, kun kyseinen piirilevy on itsekin jännitteinen. Päätehtävän lisäksi neulapetielektroniikka sisältää neljä lisäominaisuutta, jotka on myös esitetty lohkokaaviona kuvassa 14. Signaali kulkee USB–TIA/EIA-485-muuntimelta neulapedille verrattain pitkää kaapelia pitkin, joten piirille on suunniteltu transienttisuojaus sekä digitaalinen puskurointi signaalin eheyden takaamiseksi. Sammuttamalla puskuripiiri taataan neulapedin neulojen jännitteettömyys riippumatta tietokoneen lähettämästä signaalista. Kolmas ominaisuus on ohjauskortin analogisen revisiosignaalin välittäminen eteenpäin. Taajuusmuuttajan muistiin kirjoitettava ohjelmisto valitaan ohjauskortin revision perusteella ja yksi keino revision selvittämiseksi on ohjauskortilla oleva analoginen jännitesignaali, joka sisältää revisioinformaation. Tämä analoginen korkeaimpedanssinen signaali puskuroidaan ja välitetään

toiselle suunniteltavalle piirikortille. Viimeisenä lisäominaisuutena on oikosulkusuojaus. Jos ohjelmoitavassa kortissa syntyy oikosulku, virta rajoitetaan virtarajoitetulla kytkimellä kaksinkertaiseksi tavalliseen ohjelmointivirtaan verrattuna. Koska laboratorioteholähde on mitoitettu niin, että virtaa riittää viiden kortin ohjelmointiin sekä yhden hallittuun oikosulkuun, muut kortit voidaan ohjelmoida loppuun asti yhden vioittuessa.



Kuva 15: Latausaseman pääelektroniikan lohkokaavio. Lohkojen yhteydessä mainitut sivunumerot viittaavat kytkentäkaavion sivuihin liitteessä A.2.

Toinen suunniteltava osa on aseman pääelektroniikka, jonka tehtävä on kommunikoida tietokoneen kanssa ja ohjata oheislaitteita. Piirilevylle kiinnitetään erillinen *mikrokontrollerikokeilukortti (engl. microcontroller evaluation kit)*, jossa on USB-portti yhteydenpitoon tietokoneelle ja *logiikkapinnejä (GPIO, engl. General Purpose Input/Output)* muihin sähköisiin osiin liittymiseksi. USB-väylän kommunikointiin palataan mikrokontrollerin ohjelmistoa käsittelevässä luvussa 6.4. Kuvasta 15 nähdään, että ohjattavia osia on kolmea eri tyyppiä, minkä lisäksi kahdentyyppisistä osista luetaan signaali, jonka perusteella toimitaan. Tärkein ohjattava asia on edellä kuvattu neulapetielektroniikka. Mikrokontrolleri pystyy asettamaan neulapedille riittävän käyttöjännitteen, jolloin ladattava ohjelmisto voidaan siirtää tietokoneelta taajuusmuuttajaan. Neulapetielektroniikalta saadaan myös ohjelmoitavan taajuusmuuttajan revisiotieto, jonka mikrokontrolleri välittää tietokoneelle ennen ohjelmoinnin aloittamista. Toinen ohjattava laite on latausaseman neulapedin kannen lukkosolenoidi, joka estää neulapedin kannen avaamisen vahingossa, kun ohjelmiston lataus on kesken. Kanteen liittyy myös sen tilasta – auki tai kiinni – kertova mikrokytkin, joka on kahdennettu, jotta yhden komponentin hajoaminen ei vaaranna käyttäjän turvallisuutta. Käyttäjä voi päästä käsiksi vain osiin joiden käyttöjännite on 5 V, mutta suunnittelussa sovellettiin toimeksiantajayrityksen turvallisuusohjeita, jotka ovat voimassa muille samankaltaisille tuotannon sähkölaitteille. Toisaalta alhaistenkin jännitteiden poisto estää myös materiaalivahingot, jos esimerkiksi operaattorin kaulako-

ru joutuu neulapedin neuloihin oikosulkien näistä kaksi [54, s. 371]. Viimeinen ohjattava osatyyppi on latauksen tilan osoittavat valodiodit, joita on kolme jokaista ohjelmoitavaa ohjauskorttia kohden: keltainen, punainen ja vihreä. Värit kertovat latauksen olevan kesken, epäonnistunut tai valmis. Sama informaatio näytetään myös tietokoneen näytöllä, mutta jos jonkin lataus on epäonnistunut, oikean kortin poimiminen laitteesta latauksen jälkeen on helpompaa tiedon ollessa myös kortin välittömässä läheisyydessä.

Ohjelmoitavien ohjauskorttien käyttöjännite voidaan syöttää USB-keskittimen tai laboratorioteholähteen kautta. Ensimmäinen vaihtoehto toimii, mutta vaatii lisäksi toisen samanlaisen keskittimen pelkästään virransyöttöön, koska yksi USB-portti tarjoaa vain 500 mA virtaa, kun ohjelmoitaessa taajuusmuuttaja vaatii lähes 1 A virtaa. Latausaseman suunniteltava elektroniikka tarvitsee edellisestä riippumatta 5 V teholähteen. Kun ohjelmoitavien sekä suunniteltavien korttien käyttöjännite syötetään laboratorioteholähteellä, ei tarvita toista USB-keskitintä ja kokonaisuus sisältää yhden laitteen vähemmän. Pienempi rikkoutuvien osien lukumäärä parantaa kokonaisuuden luotettavuutta, jos oletetaan kokonaisuuden *vikataajuuden (engl. failure rate)* olevan sen osien vikataajuuksien summa [54, s. 387]. Huomataan tosin, että pelkkää teholähdettä käytettäessä sen tulee olla suurempitehoinen kuin jos sen rinnalla on toinen USB-keskitin virransyöttöön, joten luotettavuuden vertailu ei ole näin suoraviivaista. Kytkentäkaaviossa varataan mahdollisuus käyttää USB-portteja virran syöttöön tarvittaessa, kun asennetaan *oikosulkupalat*[20] *(engl. jumper)* piirilevylle. Pääasiallisesti käytetään laboratorioteholähdettä.

Yksittäisillä komponenteilla voidaan vaikuttaa koko latausaseman elektroniikan luotettavuuteen. Wilson [54, s. 385–386] mainitsee esimerkkinä, että kondensaattoreiden vikaantumistodennäköisyys on jopa 32-kertainen, jos niitä käytetään suurimmalla sallitulla jännitteellä verrattuna siihen, että jännite on puolet maksimista. Hän antaa ymmärtää, että myös muilla passiivisilla ja aktiivisilla komponenteilla luotettavuus kasvaa huomattavasti, mikäli niitä käytetään pienemmällä jännitteellä, virralla tai teholla kuin mitä valmistaja lupaa niiden kestävän. Komponenttien *ylimitoituksen (engl. derating)* lisäksi Wilson huomauttaa lämpötilan vaikuttavan lähes yhtä paljon niiden elinikään, sillä 10 °C nousu komponentin lämpötilassa kaksinkertaistaa komponentin vikataajuuden. Latausaseman komponentteja valittaessa noudatettiin näitä suosituksia sekä ABB:n sisäisiä mitoitusohjeita, joten komponentit ovat tietoisesti ylimitoitettuja. Lisäksi valittiin hankkijoiden työn helpottamiseksi komponentteja, joilla on vaihtoehtoisia valmistajia ja joiden valmistajat ovat ennalta hyväksyttyjä, sekä uudelleenkäytettiin samanlaisia komponentteja erilaisten komponenttien lukumäärän minimoimiseksi. Testaus ja asentajien työ huomioitiin suosimalla liittimiä, jotka voidaan johdottaa aseman ulkopuolella, ja merkitsemällä komponenttien *viittausilmaisin (engl. reference designator)*, liittimien pinnien signaalit, testipisteiden nimet ja komponenttien suunnat näkyvästi piirilevylle silkkipainatuksella [54, s. 73–74, 373–374].

Komponenttien asetteluun piirilevyllä vaikuttaa pääasiassa mekaanisesti suurten liittimien ja keskimääräistä lämpimämpien komponenttien sijoittelu, mutta lisäksi on huomioitu johtimet, reititys ja mikropiirien virransyöttö. Liittimet ovat piirilevyn reunoilla, jotta niihin päästään helposti käsiksi. Tavallista enemmän lämpenevät

[20] Oikosulkupalat R401, R402, R501, R502, R601 ja R602 on toteutettu 0 Ω vastuksilla.

komponentit ovat myös lähellä reunoja, jotta ilmavirta pääsee kulkemaan vapaasti niiden ohi kuljettaen lämpimän ilman pois [54, s. 401]. Johtimien pituuden rajoittaminen vähentää piirin tahatonta säteilyä, jota voidaan edelleen supistaa myös tekemällä käyttämättömästä kuparipinnasta maataso [58, s. 42, 146–147], sekä pienentää johdinten siirtojohtoilmiöiden vaikutusta. Toisaalta tässä sovelluksessa siirtojohtoilmiöt eivät muodostu ongelmaksi, koska signaalin etenemisnopeus tavallisessa piirilevyn kuparijohtimessa on $v \approx 152 \cdot 10^6 \, \frac{\mathrm{m}}{\mathrm{s}}$ [58, s. 151–152], joten suurimman piirissä kulkevan taajuuden $f = 1\,\mathrm{MHz}$ aallonpituus on

$$\lambda = \frac{v}{f} \approx 152\,\mathrm{m}. \tag{5}$$

Siirtojohtoilmiöt tarvitsee ottaa huomioon yleensä vain, kun aallonpituus on alle kymmenen kertaa kyseisen johtimen pituus [54, s. 34]. Nopein signaali on taajuusmuuttajalle siirrettävä datasignaali, jonka taajuuden $f = 1\,\mathrm{MHz}$ työn toimeksiantaja on määrittänyt latausasemasta riippumatta. Aiemmin kappaleessa 3.3 käsitellyn perusteella tiedetään, että Flash-muisteihin kirjoitettaessa ja niitä tyhjennettäessä muistin sisäiset operaatiot vievät eniten aikaa, joten taajuudella ei ole näissä operaatioissa suurta merkitystä. Tästä huolimatta aseman komponentit on valittu niin, että se tukee tarvittaessa jopa 10 MHz informaatiosignaalia, mikäli tulevaisuudessa muistiin kirjoitettu tieto halutaan lukea nopeammin. Johtimien reitittämisessä on pääosin käytetty leveämpiä johtimia ja suurempia välejä niiden välillä kuin mihin piirilevyn valmistaja pystyisi, jotta pienet valmistusvirheet eivät tee levystä käyttökelvotonta. Lisäksi on suosittu 45 asteen kulmia, koska syövytyskemikaali voi jäädä loukkuun suoraan kulmaan. Mikropiirien riittävä energiansaanti on varmistettu niiden välittömässä läheisyydessä olevilla erotuskondensaattoreilla. Lisäksi piirilevylle on suunniteltu panelointi, jotta sitä on helpompi käsitellä sen valmistusvaiheessa. [54, s. 51–52, 60, 243–246] Aseman sijoittelukaavio löytyy liitteestä B.

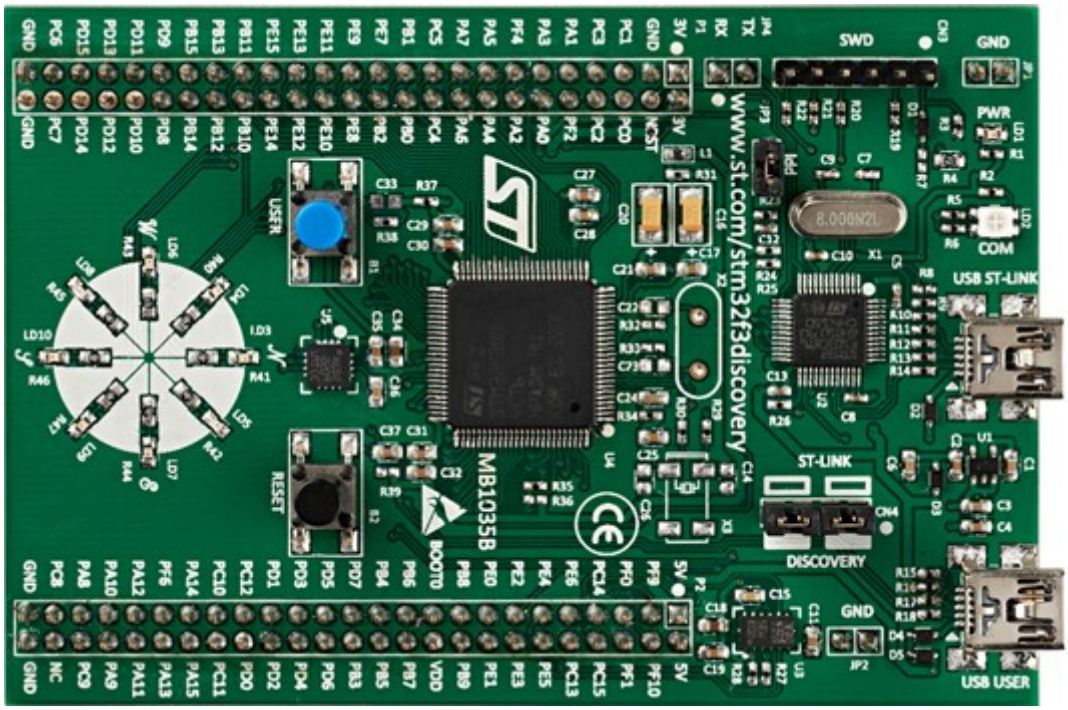
6.4 Mikrokontrolleriohjelma

Mikrokontrolleriohjelma on C-kielinen ohjelma, jota suoritetaan tauotta pääelektroniikan piirilevyyn kiinnittyvällä STMicroelectronicsin STM32 F3 Discovery *-testialustalla (engl. evaluation board tai evaluation kit)*, joka on kuvassa 16. Mikrokontrolleri toimii adapterina, joka saa käskyn tietokoneelta ja sen perusteella ohjaa elektroniikkaa. Mikrokontrolleriohjelman tärkeimmät toiminnot ovat:

1. Neulapedin kannen lukon ohjaus
2. Neulapedin kannen tilan seuraaminen
3. Neulapedin käyttöjännitteen asettaminen
4. Ohjelmoitavan kortin analogisen revisiosignaalin lukeminen
5. Ohjelmoinnin tilan kertovien valodiodien ohjaus

Ohjelma toteuttaa kaikki toiminnot mahdollisimman yksinkertaisella tavalla. Kaikki monimutkaiset algoritmit toteutetaan tietokoneella käyttöliittymäohjelmassa siten, että se

käskee mikrokontrolleria tarvittaessa. Mainitun työnjaon peruste on, että käyttöliittymän ohjelmakoodia on helpompi muokata: mikrokontrollerin uudelleenohjelmointi vaatii vähintään USB-kaapelin vaihtamista liittimestä toiseen ja kehitysympäristön asentamista. Jäljempänä selitetty käyttöliittymä ohjelmoidaan ja suoritetaan LabVIEW-ympäristössä, jossa kehitys- ja ajoympäristö ovat samassa paketissa, joten ohjelmakoodin muokkaaminen ei vaadi erillisen ohjelmiston asentamista.



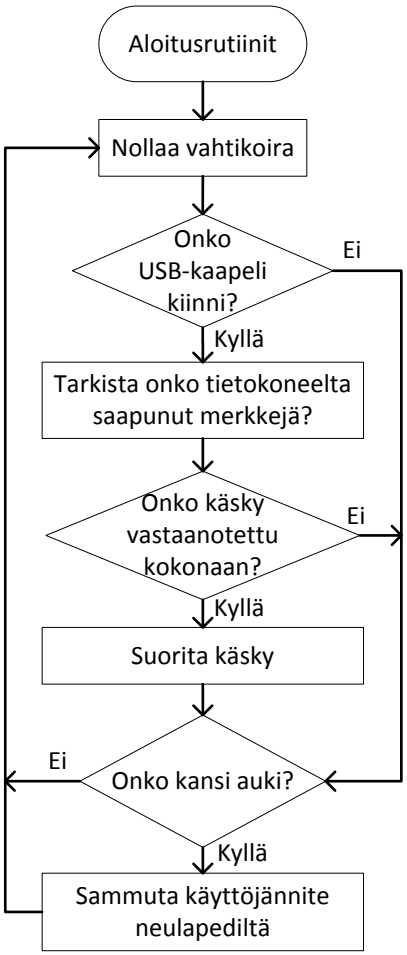
Kuva 16: STM32 F3 Discovery -testialusta. Kuva on STMicroelectronicsin Internet-sivuilta. [59]

Mikrokontrolleri kommunikoi tietokoneen kanssa USB-kaapelin avulla ja elektroniikan piirilevyjen kanssa suoraan sähköisillä logiikkasignaaleilla. Mikrokontrolleri näkyy tietokoneella virtuaalisena sarjaporttina, joten komennot voidaan lähettää ohjelmistokehityksen helpottamiseksi selkokielisenä *amerikkalaisena tiedonvaihdon standardikoodina (ASCII, engl. American Standard Code for Information Interchange)*. Lähtökohtaisesti laitteen käyttöympäristö oletetaan niin vähähäiriöiseksi, ettei bittivirheitä kommunikaatiossa esiinny. Tiedonsiirrossa on silti varauduttu häiriöihin varaamalla osa viestistä tarkistussummalle. Käytetty tarkistussumma on tyypiltään kahdeksanbittinen kahden komplementti[21], joka havaitsee yhden bitin virheen. Mikäli käyttöympäristö osoittautuu hyvin häiriöiseksi, alustava tarkistussumma voidaan korvata paremmalla uudelleenkirjoittamalla yksi funktio käyttöliittymästä ja mikrokontrolleriohjelmasta. Feldmeier on vertaillut erilaisia tarkistussummia ja niiden ominaisuuksia toisiinsa nähden sekä esittää algorit-

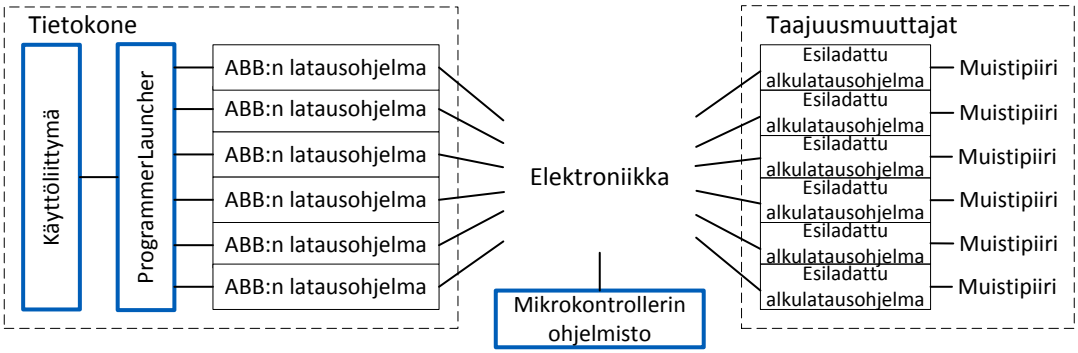
[21] Kahden komplementti -tarkistussumma määritellään seuraavasti: tietolohko on virheetön, jos alkuperäisen viestin ja tarkistussumman summa on nolla [60, s. 177]. Sen kahdeksanbittinen versio lasketaan summaamalla yksittäiset tavut (merkit) ja esittämällä summan vähiten merkitsevä tavu kahden komplementtina.

mit niiden laskemiseksi [61]. Latausasemassa viestin vastaanottavan osapuolen tulee aina varmistaa tarkistussumma ja pyytää uudelleenlähetystä, mikäli viesti on virheellinen.

Ohjelman perusrakenne on silmukka, joka odottaa käskyä ja vastaanottaessaan sen, tulkitsee ja suorittaa komennon. Yksityiskohtainen ohjelmakoodi on listattu liitteessä C. Hieman yksinkertaistettu rakenne on esitetty kuvassa 17. Alustuksen jälkeen on loputon silmukka, joka alkaa vahtikoiran nollaamisella. Jos ohjelma jää jumiin, vahtikoiraominaisuus käynnistää sen uudestaan automaattisesti [54, s. 271–274]. Sitten mikäli USB-kaapeli on yhdistetty tietokoneeseen, tarkistetaan onko tietokone lähettänyt viestin. Viesti tulkitaan ja sen sisältämä komento suoritetaan. Lopuksi varmistetaan, ettei ohjelmoitavilla korteilla ole käyttöjännitettä aseman kannen ollessa auki ja poistetaan käyttöjännite tarvittaessa. Viimeksi mainittu ominaisuus voidaan toteuttaa myös keskeytyksellä, mutta se monimutkaistaa koodia tarpeettomasti, kun huomioidaan, että käyttäjä ei pysty avaamaan kantta ja koskettamaan jännitteellisiä osia yhden silmukan kierroksen suoritusajassa, joka kestää alle kymmenen mikrosekuntia.



Kuva 17: Mikrokontrolleriohjelman vuokaavio.



Kuva 18: Eri ohjelmistojen tehtävänjako. Edellisessä luvussa käsiteltiin mikrokontrolleriohjelmistoa ja tässä luvussa käsitellään käyttöliittymää ja ProgrammerLauncher-sovellusta. Taajuusmuuttajien *alkulatausohjelma (engl. bootloader)* on hyvin pieni ohjelma, jonka avulla suoritin osaa muun muassa kirjoittaa muistipiirille. Alkulatausohjelmaa ei käsitellä tässä työssä tarkemmin.

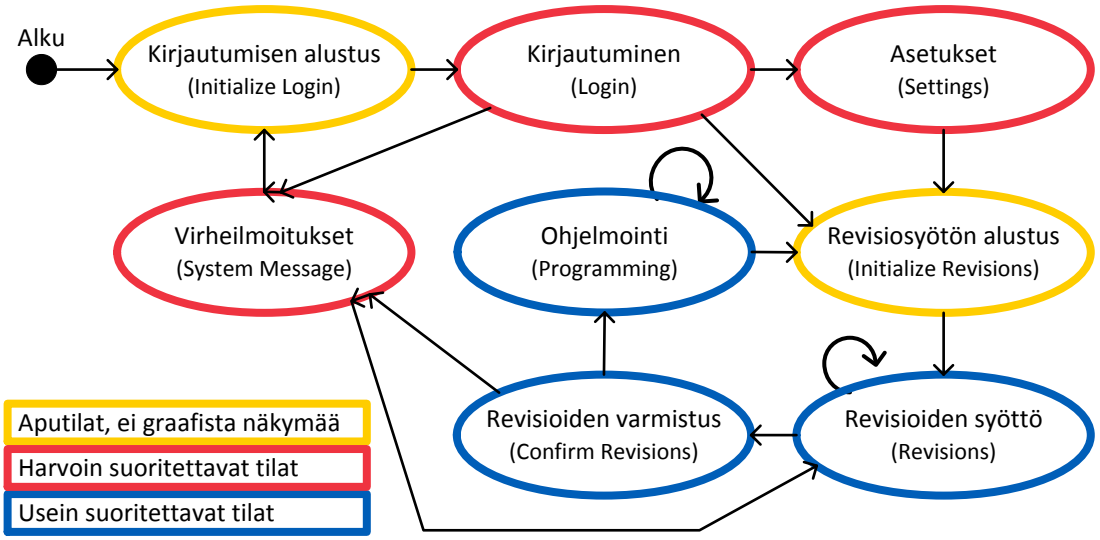
6.5 Käyttöliittymäohjelmisto

Latausaseman käyttöliittymä tietokoneella on LabVIEW-sovellus, jonka alla toimii yksittäisiä toimintoja suorittavia ohjelmia. LabVIEW on National Instrumentsin graafinen ohjelmointiympäristö, joka on tarkoitettu mittalaitteiden ohjaamiseen. LabVIEW ei tarjoa yhtä joustavia ohjelmointimahdollisuuksia kuin monet perinteiset ohjelmointikielet, kuten esimerkiksi C++, Java tai .NET. Latausaseman käyttöliittymän kannalta suurin puute on, että jos LabVIEWillä käynnistää erillisen sovelluksen, sen tulosteen pystyy lukemaan vasta erillisen sovelluksen sulkeuduttua, mikä estää ohjelmiston latauksen edistymisen seuraamisen sen ollessa käynnissä. Ongelman kiertämiseksi LabVIEWin ja varsinaisen ohjelmiston lataavan ohjelman välille kirjoitetaan lyhyt skripti, joka lukee latauksen edistymisen ja kirjoittaa sen tiedostoon, jonka LabVIEW voi lukea. Erilaisten sovellusten hierarkia on monimutkainen, joten kuva 18 esittää edellä selitetyn kuvana.

Käyttöliittymä on *tapahtumapohjainen (engl. event-driven)* tilakone. Päätiloja on kuusi, jonka lisäksi on kaksi aputilaa, joita käytetään päätilojen alustukseen, jotta graafinen ohjelmakoodi säilyy luettavana. Tilojen välillä liikutaan pääasiassa käyttäjän syötteen perusteella. Kuvassa 19 on listattu tilat ja niiden väliset siirtymät. Tavallisesti ohjelman suoritus toistaa seuraavaa kolmea tilaa:

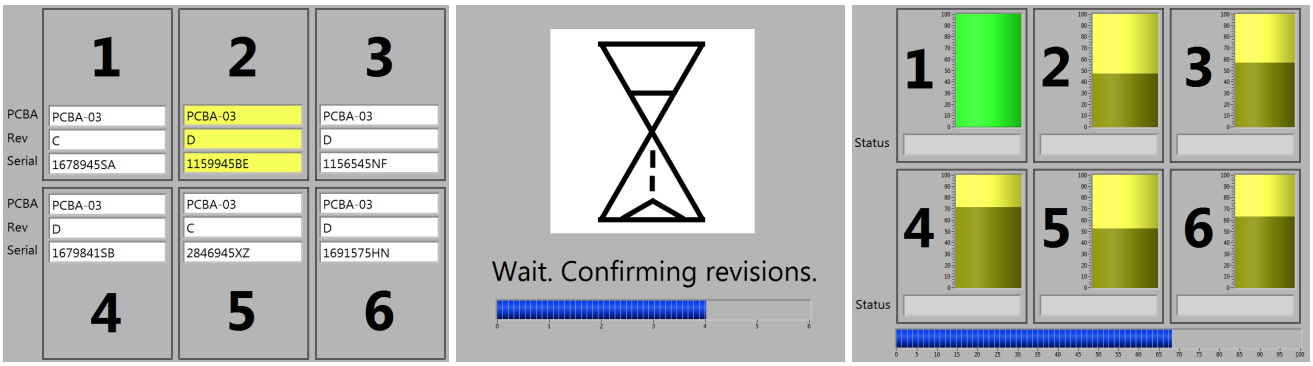
1. Revisioiden syöttö
2. Revisioiden varmistus
3. Ohjelmointi

Kuvakaappaukset näistä tiloista on kuvassa 20. Ensimmäisessä tilassa operaattori lukee jokaisen ohjelmoitavan kortin viivakoodin järjestyksessä. Seuraavassa tilassa latausaseman kansi lukitaan automaattisesti ja ohjelmoitaville korteille asetetaan käyttöjännite. Tällöin revisio saadaan myös ohjelmoitavalta kortilta analogisena jännitteenä. Jos viivakoodissa lukeva ja sähköinen revisio eroavat, operaattorin tulee



Kuva 19: Käyttöliittymän eri tilat ja niiden väliset siirtymät. Suluissa olevat tilojen nimet viittaavat liitteen D.1 käyttämiin englanninkielisiin tilojen nimiin.

ratkaista tilanne. Ohjelmointi käynnistyy heti revisioiden oikeellisuuden toteamisen jälkeen. Käyttöliittymän taustalle käynnistetään useita – yksi jokaiselle ohjelmoitavalle kortille – työn toimeksiantajan tekemiä latausohjelmia. Jokaisen kortin latauksen tilasta kerrotaan käyttäjälle edistymispalkilla. Kun kaikki lataukset ovat valmiita, eli jokainen lataus on onnistunut tai epäonnistunut, käyttöjännite poistetaan korteilta ja kannen lukko avataan. Kun operaattori avaa aseman kannen, hän voi vaihtaa kortit vielä ohjelmoimattomiin ja käyttöliittymä palaa edellä kuvatun silmukan alkuun. Tarkempi käyttöliittymän graafinen LabVIEW-lähdekoodi on liitteessä D.1.



Kuva 20: Latausaseman käyttöliittymän kolme tärkeintä tilaa: revisioiden luku, niiden varmistaminen ja varsinainen ohjelmointi

Pääsilmukan lisäksi mainitsemisen arvoista käyttöliittymässä on asemaa ylläpitävälle huoltohenkilölle asetusnäyttö sekä automaattinen lokin kirjoitus. Asetustilassa on ohje askel askeleelta, kuinka huoltohenkilö voi säätää aseman käyttöönotossa oleelliset parametrit oikein. Lisäksi kun ohjauskorteista tehdään uusia revisioita ja niitä halutaan ohjelmoida asemalla, niiden analogisten revisiosignaalien arvot tulee tallentaa

järjestelmään asetusnäytön kalibrointitoiminnolla. Loki ohjelmoinnin onnistumisesta tai epäonnistumisesta kirjoitetaan automaattisesti ABB:n käytäntöjen mukaisesti, jotta prosessia voidaan seurata ja myöhemmin analysoida.

Käyttöliittymän suunnittelussa tiedostettiin Nielsenin [62, s. 19–26] määritelmä käytettävyydestä ja siihen ohjaava heuristiikka. Määritelmän mukaan, helppokäyttöisyydellä on seuraavat viisi ominaisuutta:

1. *Opittavuus*: järjestelmä on nopea oppia työskentelyn vaatimalle tasolle.
2. *Tehokkuus*: oppimisen jälkeen järjestelmää voi käyttää tehokkaasti.
3. *Muistettavuus*: epäsäännöllinen käyttö ei vaadi järjestelmän uudelleenopettelua.
4. *Vähävirheisyys*: jos käyttäjävirheitä tapahtuu, niistä palaudutaan nopeasti.
5. *Tyytyväisyys*: käyttäjät käyttävät järjestelmää mielellään.

Näitä ominaisuuksia tavoiteltiin seuraavilla asioilla latausaseman käyttöliittymässä:

1. Ei tarpeettomia ominaisuuksia.
2. Selkeä tieto prosessin etenemisestä.
3. Käytettävissä pelkällä viivakoodinlukijalla.
4. Selkokieliset ja ratkaisuun ohjaavat virheilmoitukset.
5. Tutut symbolit painikkeille.

On kuitenkin huomattava, että tämän työn puitteissa ei ollut mahdollista testauttaa järjestelmää tulevalla loppukäyttäjällä. Nielsenin [62, s. 13–14, 192–195] mukaan käytettävyyttä mitattaessa on erittäin tärkeää haastatella nimenomaan loppukäyttäjiä, sillä järjestelmän suunnittelija tai esimiehet eivät voi korvata oikeaa käyttäjää.

ProgrammerLauncher[22] on yksinkertainen C#.NET-ohjelma, joka on erotettu LabVIEW-koodista, jotta ohjelmointiprosessin edistyksestä voidaan kertoa käyttäjälle ohjelmoinnin edetessä. Ohjelman rakenne on hyvin yksinkertainen: luetaan tiedostosta ohjelmoitavien korttien asetukset sekä edistysinformaation esitysmuoto, joka on ilmoitettu *säännöllisenä lausekkeena (engl. regular expression)*. Sen jälkeen käynnistetään varsinaisen ohjelmoinnin suorittavat ohjelmat taustalle ja aina kun niiltä saatava syöte vastaa annettua säännöllistä lauseketta, edistyksen tilaa päivitetään. Tieto siirretään LabVIEW-käyttöliittymälle erillisen tiedoston kautta, koska tiedoston lukeminen LabVIEWillä on helppoa. Jos ProgrammerLauncherista halutaan tulevaisuudessa luopua, toimeksiantajan komentoriviohjelmaa voidaan muuttaa niin, että se kirjoittaa edistyksen tiedostoon suoraan. Muutoksen on tultava ABB:n ohjelmistotiimiltä, jolla on pääsy latausohjelman lähdekoodiin. Tarkempi ProgrammerLauncherin lähdekoodi on liitteessä D.2.

[22] ProgrammerLauncher on ohjelman nimi, mutta tarkoittaa suomeksi ”Ohjelmoijien käynnistäjää”, jossa ohjelmoijalla viitataan ABB:n kehittämään yhteen taajuusmuuttajan muistiin ohjelmiston syöttävään sovellukseen.

6.6 Ylläpidettävyys ja jatkokehitysideat

Tulevaisuudessa latausasema jää jonkun henkilön tai tiimin ylläpidettäväksi. Yksi latausaseman suunnittelutavoitteista oli helppo ylläpidettävyys, joka määriteltiin luvun 6 alussa. Tässä luvussa käydään läpi ylläpitotoimenpiteet, joita voidaan odottaa tarvittavan tulevaisuudessa. Lisäksi mainitaan työn aikana nousseet jatkokehitysideat.

Mekaanisesti kuluvat osat vaativat huoltoa. Erityisesti neulapedin neulat ja kannen tilaa seuraavat mikrokytkimet tulee vaihtaa määräajoin. Tästä aiheutuva ylläpitovaiva on pieni, koska latausaseman kanssa samassa tilassa on myös muita neulapedin sisältäviä laitteita, joten kuluvat osat voidaan huoltaa yhtä aikaa muiden samankaltaisten laitteiden kanssa.

Ohjauskorteista sekä ladattavasta ohjelmistosta tulee uusia revisioita ja kokonaan uusia ohjauskorttityyppejä halutaan ohjelmoida latausasemalla. Jos uusilla piirilevyillä testipisteet ovat eri paikoissa, jokaiselle tulee teettää uusi adapteri – tai muokata vanhaa. Uudet ladattavan ohjelmiston revisiot otetaan käyttöön siirtämällä uusi ohjelma oikeaan hakemistoon tietokoneella. Molemmat edellä mainituista toimenpiteistä vaativat asetustiedostojen muokkaamista, mutta latausaseman ohjelmistojen lähdekoodia ei tarvitse muokata.

Jatkokehitykseksi on esitetty aseman laajentamista tukemaan kuuden ohjauskortin sijaan 12:ta pienikokoista ohjauskorttia sekä pelkkien piirilevyjen ohjelmoinnin sijaan valmiiden taajuusmuuttajien ohjelmiston päivittämistä. Molemmissa kehitysideoissa laitteistoa pitää moninkertaistaa tai jättää pois, mutta ei suunnitella mitään uutta. Ohjelmista ainoastaan graafista käyttöliittymää tarvitsee muokata. Muutokset ovat yksinkertaisia, joten jatkokehitysehdotukset ovat toteutettavissa pienellä työmäärällä.

Latausasemasta on kirjoitettu työn toimeksiantajalle yksityiskohtainen dokumentaatio. Se kattaa muun muassa seuraavat asiat:

- Elektroniikan valmistus
- Järjestelmän asennus
- Käyttöönotto
- Käyttöohje
- Ylläpito

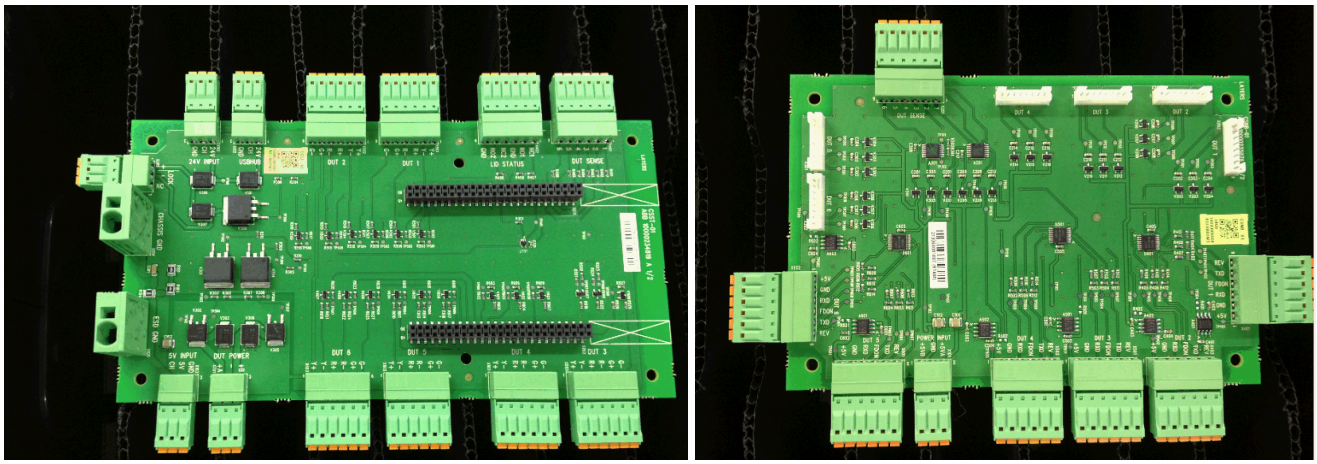
Näiden sisältö ja laajuus ovat soveltuvin osin samat kuin toimeksiantajan muissakin tuotantolaitteissa. Toimeksiantaja on tarkistanut, että dokumentaatio kattaa latausaseman käyttöönoton ja mahdollisen jatkokehityksen riittävällä tarkkuudella.

Ylläpidosta myöhemmin aiheutuvaa todellista työmäärää ja kustannuksia ei voida todeta ennen latausaseman käyttöönottoa. Voidaan kuitenkin arvioida, että ylläpito ei aiheuta millekään taholle merkittävästi lisätyötä. Jos ylläpidosta siirrytään jatkokehittämiseen, suunnitteluresursseja tarvitaan, mutta silloinkin vain vähän. Latausasema on kattavasti dokumentoitu, joten oletettavasti ongelmatilanteisiin löytyy ratkaisu, mutta myös tämä on vaikea todeta ennen kuin latausasema on otettu käyttöön.

7 Tutkimustulokset

Latausaseman elektroniikan suunnittelun ja valmistuksen jälkeen se testattiin yhdessä asemaa varten suunniteltujen ohjelmistojen kanssa. Valmiit piirikortit ovat kuvassa 21. Testitulokset on esitetty seuraavassa, minkä jälkeen tarkastellaan, miten kokonaisuus täyttää sille kappaleessa 5 asetetut vaatimukset. Työn mekaniikka valmistettiin lähellä tehdasta, jossa latausasema tullaan ottamaan ensimmäisenä käyttöön, joten lopulliseen mekaniikkaan ei ollut mahdollista tutustua.

Työn liitteenä olevien suunnitteludokumenttien suhde tässä luvussa esitettyihin korjauksiin riippuu kaavion tyypistä. Elektroniikkakaavioita liitteissä A–B ei muutettu myöhemmin tässä luvussa esitettyjen käsin tehtävien muutosten pohjalta, vaan liitteenä ovat tiedostot, joiden perusteella piirilevyt valmistettiin. Ohjelmiston lähdekoodi liitteissä C–D on testien jälkeinen korjattu versio.



Kuva 21: Vasemmalla on latausaseman pääelektroniikka ja oikealla on neulapetielektroniikka. Pääelektroniikan kahteen pitkään liittimeen kiinnitetään kuvan 16 mikrokontrolleripiirikortti. Kuvat on ottanut piirilevyt valmistanut ABB:n alihankkija heti komponenttien latomisen jälkeen.

7.1 Elektroniikan ja ohjelmiston testitulokset

Elektroniikan ja ohjelmiston testauksella tarkoitetaan tässä yhteydessä niiden toimivuuden varmistamista. Elektroniikasta tarkastetaan, että kaikki suunnitellut ominaisuudet toimivat ja mitataan useiden rajapintojen ja kaapeleiden vaikutus signaalin eheyteen, koska se herätti keskustelua suunnitteluvaiheessa. Ohjelmistojen toiminta varmistetaan käymällä ohjelmiston ominaisuudet läpi. Lopuksi todellista käyttötilannetta jäljitellään kirjoittamalla ohjelmisto toistuvasti taajuusmuuttajien ohjauskorteille.

Taulukko 5: Elektroniikan testauksen laitteisto kuvan 13 mukaisesti ryhmiteltynä

Laite	Malli
Virransyöttö:	
$5\,\mathrm{V_{dc}}$ laboratoriovirtalähde	PS3010L
$24\,\mathrm{V_{dc}}$ laboratoriovirtalähde	TTi EL302T
Signaalin polku:	
Tietokone	Lenovo T430
USB-keskitin	MOXA UPort 407
USB–TIA/EIA-485-muunnin	6 kpl, ABB:n sisäinen malli
Taajuusmuuttajan ohjauskortit	– Latausaseman tukemia *pienempiä* ohjauskortteja
	2 kpl, revisio D
	4 kpl, revisio E
	– Latausaseman tukemia *suurempia* ohjauskortteja
	6 kpl, revisio E
Muuta:	
Solenoidi	Pontiac F0413A
Mikrokytkimet	2 kpl, Omron SS-5GLT
Valodiodit	6 kpl, Lumex Opto SSI-LXMP5011GC-150
	6 kpl, Lumex Opto SSI-LXMP5011SRC-150
	6 kpl, Lumex Opto SSI-LXMP5011YC-150
Oskilloskooppi	Tektronix DPO 4034
Yleismittari	Agilent U1233A

Taulukko 6: Elektroniikkaan suunniteltujen ominaisuuksien testitulokset

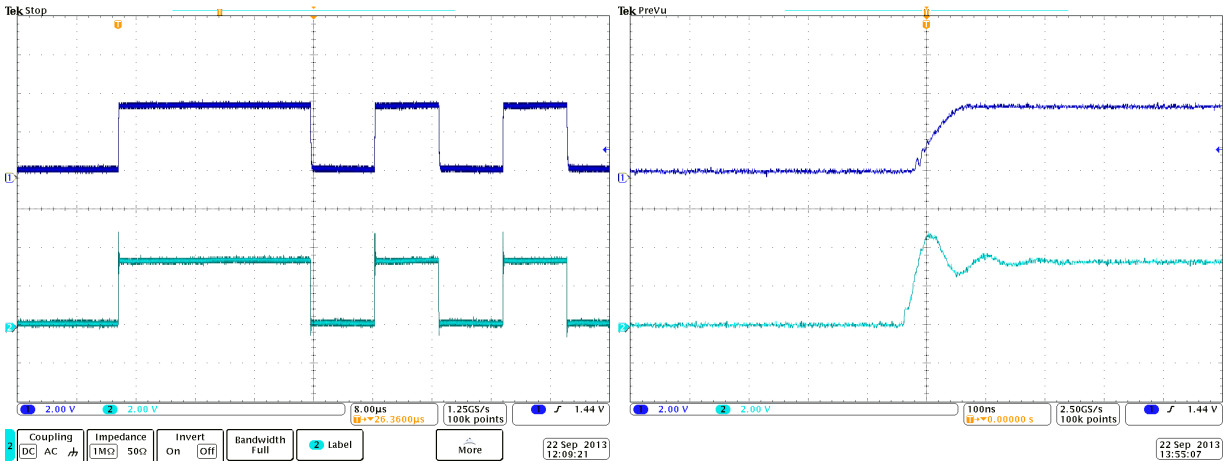
Ominaisuus	Oikean toiminnan kriteeri	Lopputulos
Pääelektroniikka:		
Valodiodit	Mikrokontrolleri ohjaa oikeaa valodiodia komennosta.	Toimii
	Eri värien kirkkaus on silmämääräisesti samankaltainen.	Toimii
Mikrokytkimet	Mikrokontrolleri voi lukea yksittäisten kytkinten tilan.	Toimii
	Neulapedin käyttöjännite katkeaa, kun kytkin on auki.	Toimii
Lukkosolenoidi	Mikrokontrolleri ohjaa solenoidin tilaa komennosta.	Korjaten
Neulapedin käyttöjännite	Mikrokontrolleri ohjaa käyttöjännitettä komennosta.	Korjaten
Neulapetielektroniikka:		
Signaalin välitys	TIA/EIA-485-muotoinen signaali pysyy muuttumattomana.	Toimii
Revision lukeminen	Ohjauskortin revisio voidaan lukea sähköisesti.	Osittain
Oikosulkusuojaus	Ohjauskortin oikosulku ei katkaise muiden käyttöjännitettä.	Toimii[23]

[23]Testikäyttöön varattuja ohjauskortteja oli hyvin vähän ja niiden mahdollinen rikkoontuminen haluttiin välttää, joten testi suoritettiin seuraavasti: yhden ohjauskorttipaikan käyttöjännite ja maa yhdistettiin johdolla ja muihin ohjauskorttipaikkoihin asetettiin 8,2 Ω vastus simuloimaan ohjauskortin keskimäärin käyttämää virtaa. Neulapetielektroniikan käyttöjännite kytkettiin päälle ja tarkistettiin, että käyttöjännite pysyy vakiona ei-oikosuljetuilla ohjauskorttipaikoilla ja että laboratoriovirtalähde jaksaa tarjota riittävästi virtaa.

Latausaseman elektroniikkaa ja ohjelmistoa, jotka kuvattiin luvuissa 6.3–6.5, testattiin työn toimeksiantajan tiloissa. Testattu järjestelmä eroaa kuvasta 13 siten, että *varavirtalähdettä (UPS, engl. Uninterruptable Power Supply)* ja neulapetiä ei ollut käytettävissä. Neulapeti korvattiin juottamalla 30 cm johdot neulapetielektroniikasta testattavan taajuusmuuttajan ohjauskortin testipisteisiin. Johtojen pituus vastaa todellisuutta, sillä vaikka latausaseman sisällä johdinten pituudet pyritään minimoimaan, ohjauskorttien fyysisen koon vuoksi neulapetielektroniikasta kauimpana oleviin ohjauskortteihin etäisyys tulee olemaan vähintään 20 cm. Testeissä käytetty laitteisto on lueteltu taulukossa 5. Lopullisessa latausasemassa saatetaan käyttää aavistuksen eri laitteistoa saatavuuden mukaan. Testattava elektroniikka ja ABB:n toimittamat osat pysyvät kuitenkin samoina.

Elektroniikan yksittäisten ominaisuuksien toimivaksi toteamisen kriteerit voivat olla monitulkintaisia joissain tapauksissa, joten taulukkoon 6 on kirjattu sekä testikriteerit että niiden lopputulokset. Pääosin ominaisuudet toimivat. Toimimattomien ominaisuuksiin korjaamiseen sekä osittain toimivaan ominaisuuteen palataan myöhemmin tässä luvussa.

Suunnitteluvaiheessa pohdittiin ylimääräisten sähkömekaanisten rajapintojen ja kaapelien pituuden vaikutusta järjestelmään. Etäisyys USB–TIA/EIA-485-muuntimelta ohjauskortin testipisteille on noin 60 cm ja neulapetielektroniikka lisää tähän kaksi liitinrajapintaa, joita ei ollut aiemmin. Kuvasta 22 nähdään, että ohjelmointiin käytetyillä taajuuksilla signaalin muoto säilyy hyvin. Kun signaalia tarkastellaan hyvin läheltä, nähdään neulapetielektroniikan aavistuksen hidastavan sen nousunopeutta mutta toisaalta myös poistavan USB–TIA/EIA-485-muuntimen ulostulossa esiintyvän lyhyen *jännitepiikin (engl. overshoot)*. Signaalien erojen ollessa hyvin pieniä, voidaan todeta sisääntulevan signaalin pysyvän muuttumattomana pidemmistä johdoista ja lisärajapinnoista huolimatta käytetyillä taajuuksilla.



Kuva 22: Ylemmässä kanavassa on ohjauskortin testipisteiltä mitattu signaali ja alemmassa kanavassa on USB–TIA/EIA-485-muuntimen ulostulosta mitattu signaali.

Pääelektroniikkapiirilevy oli suunniteltu hallitsemaan sekä lukkosolenoidia että ohjelmoitavien ohjauskorttien käyttöjännitettä logiikkatasoilla ohjattavilla N-tyypin kanavatransistoreilla[24], mutta kanavatransistorit eivät siirtyneet johtavaan tilaan 3 V hilajän-

[24]Piirikaaviossa liitteessä A.1 kyseiset komponentit on nimetty V301, V304, V308.

nitteellä. Niille oli valittu neljä vaihtoehtoista komponenttia eri valmistajilta, joista piirilevyn valmistanut alihankkija sai latoa helpoiten saatavilla olevan. Datalehtien tarkemman lukemisen jälkeen selvisi, että yksi, joka oli päädytty latomaan kortille, osoittautui tarkoitukseen sopimattomaksi. Ladottu N-tyypin kanavatransistori rupesi johtamaan datalehden mukaan 2,0–4,0 V hilajännitteellä, kun tarkoitus oli käyttää viimeistään 2,5 V hilajännitteellä johtavaa N-tyypin kanavatransistoria. Kun vääräntyyppiset kanavatransistorit vaihdettiin oikeanlaisiin, ominaisuudet toimivat ongelmitta. Komponenttien vaihto tehtiin käsin juottamalla.

Revisioiden lukuominaisuus toimi vain osittain. Alun perin tarkoitus oli, että ohjelmoitavat taajuusmuuttajan ohjauskortit kertovat revisionsa tietyntasoisena tasajännitesignaalina, joka luetaan mikrokontrollerilla. Testatessa huomattiin, että suuremman ohjauskortin revisiosignaali on pois päältä, kun käytetään ainoastaan 5 V käyttöjännitettä. Tämän vuoksi suuremman ohjauskortin revisio voidaan lukea ainoastaan sen viivakoodista. Ohjelmoitaessa pienempiä ohjauskortteja, niiden revisio luetaan ensin niiden viivakoodista, jonka jälkeen se varmistetaan vielä sähköisestä signaalista.

Koska ohjelmistossa on hyvin rajattu määrä ominaisuuksia, ohjelmiston toimivuus testattiin käymällä kaikki läpi. Testi suoritettiin seuraavassa järjestyksessä:

1. Kutsuttiin kaikkia mikrokontrollerin tukemia komentoja.
2. Tehtiin ohjelmiston käyttöönotto huoltohenkilön käyttöliittymän avulla.
3. Ohjelmoitiin pienempiä ohjauskortteja operaattorin käyttöliittymän avulla.
4. Testattiin kaikki virhetilanteet.

Ohjelmointivirheiden ilmetessä ne korjattiin ja testi uusittiin. Suunnitellusta poikettiin kahdella uudella ominaisuudella: latausohjelmien käynnistysviiveellä ja välittömällä uudelleenyrityksellä. Testeissä ilmeni kaksi tapausta, jolloin lataus voi epäonnistua. Ensimmäisenä jos latausohjelma käynnistetään välittömästi ohjauskortille käyttöjännite asetettaessa, sen suoritin ei ole ehtinyt välttämättä vielä suorittaa käynnistysrutiinejaan loppuun eikä siihen saada yhteyttä. Toisena jos rinnakkain ajettavat latausohjelmat käynnistetään ajallisesti liian lähellä toisiaan, lataus epäonnistuu välittömästi. Virheilmoituksen perusteella tämä johtuu siitä, että ne pyrkivät käyttämään samoja resursseja. Näiden syiden vuoksi lisättiin viive aina ennen jokaisen latausohjelman käynnistämistä. Testeissä käytettiin viiveenä viittä sekuntia, jolloin lataus toimii luotettavasti. Seuraavassa kuvatussa rasitustestissä ilmeni, että noin 3 % todennäköisyydellä ohjauskorttiin ei saatu yhteyttä heti latauksen alettua edes viiveominaisuuden kanssa. Ilmiö on olemassa myös ilman latausasemaa, joskaan sen ilmenemistodennäköisyydestä ei ole tilastoitua tietoa. Ongelman kiertämiseksi ohjelmistoon lisättiin ominaisuus, joka yrittää käynnistää epäonnistuneen kortin latauksen uudestaan, jos sen lataus epäonnistuu välittömästi sitä yritettäessä. Ominaisuuden vuoksi kokonaislatausaika pitenee korkeintaan sekunneilla.

Kokonaisuutta testattiin rasitustestillä. Oikealla tuotantolinjalla ohjelmoidaan päivässä satoja ohjauskortteja, mutta koska aseman tukemia ohjauskortteja ei vielä valmisteta massana, aseman testaamista varten oli saatavilla ainoastaan kuusi kappaletta molempia tuettuja ohjauskortteja. Latausaseman hyödyntämä ABB:n komentorivipohjainen latauskäyttöinen työkalu ei myöskään ollut lopullinen versio, jota tullaan käyttämään massatuotannossa.

Saatavilla olleella ohjelmistoversiolla lataukseen kului noin 13 minuuttia. Tuotantolinjaa simuloitiin kirjoittamalla ohjelmisto pienemmille ohjauskorteille kymmenen kertaa peräkkäin. Huomattiin, että kahdella kymmenestä ohjelmointikierroksesta ei saatu yhteyttä yhteen ohjauskorttiin, joka oli eri molemmilla kerroilla, heti latauksen alussa. Tämän vuoksi latausaseman ohjelmistoon tehtiin edellä kuvattu lisäominaisuus, jolla latausta yritetään uudelleen, jos se epäonnistuu heti latauksen alussa. Ominaisuuden lisäämisen jälkeen uudessa testissä epäonnistumista ei havaittu. Kuuden ohjauskortin ohjelmointiin kului aikaa joka kierroksella vähintään 13 minuuttia ja 25 sekuntia ja korkeintaan 14 minuuttia ja 30 sekuntia. Latausaika on hyvin lähellä yksittäisen ohjauskortin vastaavaa, joten voidaan todeta latausaseman ohjelmoivan kuusi ohjauskorttia lähes yhtä nopeasti kuin yksittäinen ohjauskortti voidaan ohjelmoida.

Testattaessa suurempia ohjauskortteja huomattiin, että niiden lataus epäonnistuu keskimäärin noin 50 % todennäköisyydellä myös ilman latausasemaa. Asiaa tarkemmin selvitettäessä ilmeni, että ongelma liittyy oletettavasti nykyiseen ohjelmistoversioon ja sen ratkaisu odottaa ohjelmistotiimin toimia. Suuremmilla ohjauskorteilla ei suoritettu rasitustestiä, koska nykyisellä ohjelmistoversiolla epäonnistumisprosentti on niin suuri. Testattiin kuitenkin yksittäisten suurempien ohjauskorttien ohjelmointia ja todettiin, ettei epäonnistumistodennäköisyys muuttunut latausasemaa käytettäessä, kun otos oli kymmenen latausyritystä.

Latausaseman elektroniikka ja ohjelmisto toimivat siis pienten korjausten jälkeen pienemmälle ohjauskortille. Suuremman kanssa on ongelmia, jotka eivät liity latausasemaa varten suunniteltuihin osiin, mutta liittyvät silti latausasemakokonaisuuteen, koska kyseisiä ohjauskortteja ei voi toistaiseksi ohjelmoida asemalla. Testien perusteella aseman toimivuus on siis osittainen, mutta voidaan olettaa, että latausasema tulee toimimaan kokonaisuudessaan ABB:n komentorivityökalun päivityksen myötä. Tämän toteaminen jää diplomityön ulkopuolelle, koska päivityksen aikataulusta ei ole varmuutta.

7.2 Asetettujen vaatimuksien toteutuminen

Luvussa 5 lueteltiin toimeksiantajan vaatimukset latausasemalle. Seuraavassa on lueteltu jokainen vaatimus ja kommentoitu siinä onnistumista.

1. *Kortit helposti liitettäviä*: Suunnitelmien perusteella kortit ovat helposti liitettäviä, mutta koska lopulliseen mekaniikkaan ei päästy tutustumaan, tätä ei voida kommentoida enempää.

2. *Helposti opittava*: Loppukäyttäjiä ei ollut mahdollista tavata projektin puitteissa, mutta kaikki kenelle latausaseman prototyyppiä esiteltiin, ymmärsivät käyttölogiikan operaattorin näkökulmasta jo ensimmäisellä esityskerralla. Todetaan tämä vaatimus täytetyksi.

3. *Revisio määritettävissä*: Latausaseman kaikki ohjelmistoversiot sisältävät yksilöllisen versionumeron, joita voidaan tarkastella kootusti yhdestä käyttöliittymän näkymästä. Todetaan tämä vaatimus täytetyksi.

4. *Tuki useille ohjauskorteille*: Latausasema toistaa TIA/EIA-485-muotoisen ohjelmointisignaalin sellaisenaan ohjauskortista riippumatta. Tämä varmistettiin pienemmillä tuettavilla ohjauskorteilla. Suuremmilla ohjauskorteilla tätä ei kuitenkaan voitu osoittaa niiden latausohjelman nykyisestä ohjelmistoversiosta johtuen. Todetaan vaatimus täytetyksi varauksella: latausasema ei välitä mikä ohjauskortti sisällä on, mutta tämän työn puitteissa useaa eri ohjauskorttityyppiä ei voitu testata täysin kattavasti.

5. *Tuki useille korttien revisioille*: Latausasema valitsee ladattavan ohjelmiston ohjauskortin tyypin ja revision perusteella, mikä tarkoittaa, että eri tyyppisiä ja revisioisia ohjauskortteja voidaan ohjelmoida yhtäaikaisesti. Pienempien ohjauskorttien testit suoritettiin kahdella eri revisiolla onnistuneesti. Todetaan tämä vaatimus täytetyksi.

6. *Toiminnan jäljitettävyys*: Lokitiedosto ohjelmoiduista ja epäonnistuneista korteista kirjoitetaan toimeksiantajan määritysten mukaisesti. Todetaan tämä vaatimus täytetyksi.

Yleisesti todetaan latausaseman suunnittelu onnistuneeksi. Huomataan kuitenkin, että suuremman ohjauskortin testaus jäi kesken sen latausohjelman nykyisen version ollessa epäluotettava. On todennäköistä, että latausasema tulee toimimaan täysin ohjelmistopäivityksen jälkeen, mutta tämän seuraaminen jää työn ulkopuolelle.

8 Johtopäätökset ja yhteenveto

Diplomityön aikana muodostuneet johtopäätökset luetellaan seuraavassa. Luku on jaettu kahteen osuuteen. Ensin otetaan kantaa siihen, voisiko nopeasti ohjelmoitavia, mutta vähemmän luotettavia NAND-tyyppisiä Flash-muisteja käyttää pitkäikäisissä teollisuussovelluksissa. Sen jälkeen arvioidaan, onnistuiko latausasema, jolla ainakin NOR-Flashiä käyttävät uuden sukupolven taajuusmuuttajat tullaan ohjelmoimaan, ja kuinka paljon työn toimeksiantaja lopulta hyötyy siitä.

8.1 Flash-muisti

Flash-muisteja on helposti saatavilla pääasiassa kahta eri tyyppiä: NAND- ja NOR-muisteja. Helppo saatavuus on tärkeä komponentin valintakriteeri muistipiirin sisältävissä laajemmissa järjestelmissä, koska jos yhden muistipiirin valmistus lopetetaan, valmistajan tahdosta tai esimerkiksi luonnonkatastrofin vuoksi, yleiselle komponentille on suurempi todennäköisyys löytää korvaava komponentti kuin harvinaiselle. Molemmilla muistityypeistä on niille ominaisia vahvuuksia ja heikkouksia.

NOR-muisteja on käytetty perinteisesti niiden nopean satunnaisosoituksen sekä luotettavuuden vuoksi. Nopean satunnaisosoituksen avulla ohjelmisto voidaan suorittaa suoraan muistipiiriltä ilman sen lataamista haihtuvaan muistiin. Tämä ominaisuus ei siedä yhtäkään bittivirhettä, mikä osaltaan on ohjannut muistivalmistajia pitämään NORien luotettavuuden vahvana [9, s. 40]. Valmistajat takaavat 100 % virheettömyyden NOR-muisteille, minkä vuoksi ne eivät tarvitse virheenkorjausta. NOR-muisteja ei ole tarkoitettu suuren tietomäärän varastointiin ja lisäksi NOR-solu on NAND-solua suurempi, joten niitä on saatavilla pääosin NANDeja pienemmillä muistikapasiteeteilla.

NAND-muistit on alun perin suunniteltu massamuisteiksi. Sen vuoksi niiden tallennuskapasiteetti on tavallisesti suuri. Lisäksi ne ovat NOReja useammin monitasoisia, eli yhteen muistisoluun tallennetaan enemmän kuin yksi bitti informaatiota. NANDien kirjoitus- ja tyhjennysoperaatiot aikayksikköä kohden ovat huomattavasti NOReja nopeampia, koska operaatiot suoritetaan enemmän rinnakkain. Lukunopeus on molemmissa samaa suuruusluokkaa. NANDien satunnaisosoitusnopeus on NORia heikompi, mutta jos käytetään lisäksi haihtuvaa muistia välimuistina, voidaan ohjelmakoodia suorittaa NORiin verrattavalla nopeudella [17]. NAND-muisteilla voi olla jo valmistettaessa epäluotettavia soluja ja niitä voi syntyä myöhemmin käytön aikana. NAND-muisteja käytettäessä tulee varmistua, että järjestelmässä on tavalliset muistiohjaimen toiminnot: virheenkorjaus, kulutuksen tasaus, roskienkeräys sekä viallisten lohkojen hallinta.

Flash-muistien kirjoitus- ja tyhjennysaikojen muodostuminen voitiin selittää konseptitasolla, mutta absoluuttisiin kestoihin ei saatu selvää vastausta. Elektroniikkasuunnittelijan kannalta kirjoitus- ja tyhjennysajoissa mielenkiintoista on, että muistin tilaa muokkaavien operaatioiden maksimikesto voi olla jopa kymmeniä kertoja tyypillistä aikaa pidempi [32]. Tämä johtuu siitä, että nämä operaatiot muokkaavat muistin tilaa hetken ja sitten tarkistavat tuloksen. Jos tulos ei ollut haluttu, silmukkaa toistetaan tiettyyn rajaan asti niin monta kertaa, että saadaan oikea lopputulos. Hidastaviksi tekijöiksi datalehdissä mainittiin yleisesti muistipiirin käyttö sille luvattujen lämpötila-, käyttöjännite- ja tyhjennyskerta-arvojen rajoilla. Toisin sanoen, muistin kirjoitus- ja tyhjennysoperaa-

tiot ovat nopeampia uudelle muistikomponentille huoneenlämmössä sallittujen käyttöjännitearvojen keskellä. Teorian perusteella erityisesti muistisolun oksidin huonontuminen vaikuttaa tarvittavien muistin tilaa muokkaavan algoritmin kierrosten lukumäärään ja täten koko operaation kestoon. Operaatioiden tarkemmasta suorituksesta on kirjoitettu matemaattisia kaavoja myöten, mutta ilman esimerkkilukuarvoja [8, s. 55–88][9, s. 129–178][10, s. 35–70]. Tuorein julkaisu, joka jakaa operaatiot vielä pienempiin osiin ja mainitsee niiden osien kestoja lukuina, on vuodelta 1995 [13]. Julkaisussa mainittujen lukuarvojen summa vastaa nykyistenkin muistipiirien datalehdissä mainittuja arvoja, mutta koska julkaisusta on jo 18 vuotta, sen yhdistämiseen nykykomponentteihin on syytä suhtautua varauksella.

Oleelliset luotettavuusnäkökohdat keskittyvät Flash-muisteilla käytännössä kokonaan yhden teeman ympärille: muistisolussa olevan oksidin huonontumiseen, mikä kiihdyttää muistisolussa olevan varauksen tahatonta muuttumista. Oksidin huonontuminen tarkoittaa sitä, että kirjoitettaessa ja tyhjennettäessä, oksidiin ja sen välittömään ympäristöön tarrautuu elektroneja, jotka kasautuessaan vaikuttavat muistisolun kynnysjännitteeseen. Flash-muisteilla on monia vikaantumismekanismeja, joista vähintään kaikki yleisimmät ovat verrannollisia tyhjennyskertojen lukumäärään, toisin sanoen oksidin kuntoon. Solun varaus voi muuttua tahattomasti kumpaan tahansa suuntaan. Dominoiva vikaantumismekanismi erityisesti monitasoisilla muisteilla liittyy selkeästi tiedon säilyvyyteen, eli että ajan kuluessa kirjoitetun tiedon sisältö muuttuu. Toissijaisina ongelmakohtina ovat *luku- ja kirjoitushäiriöt (engl. read and write disturb)*, mutta niiden todennäköisyys on huomattavasti, jopa 3–4 dekadia, matalampi kuin säilyvyysvirheellä. [49][51] Yksitasoisten NANDien ja ylipäätään NORien luotettavuusongelmien keskinäiseen järjestykseen ja suuruusluokkiin liittyen ei löytynyt yhtä laadukasta tutkittua tietoa. Jos kuitenkin huomioidaan monitasoisuuden teoreettinen vaikutus ja NANDien ja NORien yhtäläisyydet, voidaan perustellusti olettaa suurimpien tekijöiden olevan samoja kuin edellä mainituissa tutkimuksissa.

Tärkeimpiin luotettavuusongelmiin on olemassa toimivia ratkaisuja. Koska kirjoitus- ja tyhjennyskertojen lukumäärä kiihdyttää tavallisimpia vikaantumismenetelmiä, on tärkeää kirjoittaa tasaisesti kaikkiin soluihin siten, että minkään yksittäisen solun kirjoitus- ja tyhjennyskertojen lukumäärä ei ole huomattavasti keskiarvoa korkeampi. Tästä näkökulmasta kulutuksen tasaus on hyödyllinen ominaisuus sekä NAND- että NOR-muisteissa. Tiedon säilyvyyteen auttaa huomattavasti matala käyttölämpötila sekä tiedon uudelleenkirjoitus, eli virkistys, säännöllisin väliajoin. Jo nyt yksisoluisille muisteille luvataan tavallisesti 10 tai 20 vuoden säilyvyys, joskin NANDien tapauksessa luvattu aika vaatii virheenkorjauksen käyttöä. Säilyvyysaika voidaan moninkertaistaa virkistämällä tieto tai käyttämällä muistia matalammassa lämpötilassa [43][45]. On huomattavaa, että monitasoisilla soluilla varauksen arvon on oltava tarkemmin rajatulla alueella kuin yksitasoisilla soluilla, joten valmistajat lupaavat monitasoisia soluja käyttäville muisteille huomattavasti vähemmän tyhjennyskertoja ennen kuin luotettavuus heikkenee kestämättömäksi. Kirjoitus- ja lukuhäiriöiden estämiseksi on syytä harkita virheenkorjauksen käyttämistä. NAND-muistien datalehdissä valmistajat suosittelevat sitä käytettäväksi joka tapauksessa ja jotkin NAND-rajapinnat tukevat automaattista virheenkorjausta siten, että isäntäjärjestelmän ei tarvitse huolehtia siitä.

Tasaiset käyttöjännitteet sallittujen rajojen keskellä sekä oikea käynnistys- ja sammu-

tusmenettely ovat tärkeitä suosituksia. Muistivalmistajien dokumentaation mukaan käynnistyksessä tulee odottaa riittävän pitkään, että käyttöjännite on noussut oikealle tasolle ja muistipiiri on suorittanut sisäiset käynnistysrutiininsa. Tässä kestää kymmeniä mikrosekunteja. Sammutuksessa tulee huolehtia ohjauspinnien asettamisesta oikeaan jännitetasoon. Mahdollisuuksien mukaan on hyödyllistä myös huolehtia siitä, ettei muistipiirin käyttöjännite putoa muistia käytettäessä, koska se voi aiheuttaa vasta myöhemmin ilmeneviä luotettavuusongelmia [42].

Flash-muisteilla ei ole yhtenäistä rajapintaa ajurimielessä, mutta erilaisten rajapintojen lukumäärä on hillitty. Jos muistit ovat ratkaisevasti erilaisia, esimerkiksi sarjamuotoisesti ja rinnakkaisesti käytettävät NOR-muistit, niiden käyttö on hyvin erilaista. Kuitenkin yhden tällaisen NOR-ryhmän sisällä muistit toimivat hyvin samankaltaisesti. NAND-tyypillä erilaisia rajapintoja on useampia, mutta toisaalta ne voivat myös sisältää parhaimmillaan kokonaisen muistiohjaimen, jolloin muistipiiriä käyttävän sovelluksen ei tarvitse huolehtia muistin yksityiskohdista. Kaikille Flash-tyypeille yhteisen ajurin toteuttaminen on selkeästi äärimmäisen työläs tehtävä, mutta sen sijaan jos valitaan jokin tietty rajapinta, sille voidaan kirjoittaa ajuri ja on mahdollista löytää vaihtoehtoisia muistipiirejä, jotka toimivat suoraan ensisijaisen tilalla.

Nykyään työn toimeksiantaja käyttää sovelluksissaan NOR-tyyppisiä muisteja. Jos niiden kapasiteetti on riittävä, ja kirjoitus- sekä tyhjennysnopeus tyydyttävät, tai jos ohjelmakoodia halutaan erityisesti ajaa suoraan muistilta, NOR-muistit ovat hyvä valinta. Tämän työn pohjalta kuitenkin todetaan, että jos edellä mainitut NORin ominaisuudet ovat rajoitteita, NAND-muistit ovat hyvä vaihtoehto. Jos oletetaan ABB:n käyttävän aiemmissa luvuissa käsiteltyjä 64 Mb NOR-muistipiirejä, niiden tyhjentämiseen ja uudelleenkirjoittamiseen kuluu noin minuutti aikaa parhaassakin tapauksessa. NANDeillä aikaa kuluu yksittäisiä sekunteja. NANDien tieto saadaan säilymään useita kymmeniä vuosia, kun se virkistetään välillä, esimerkiksi kerran kymmenessä vuodessa tai hieman useammin. Virkistyksen toteutus riippuu siitä, käytetäänkö itse toteutettua muistiohjainta vai valmista sellaista, esimerkiksi muistikortin sisällä olevaa. Jos muistiohjain tehdään itse, virkistys tarkoittaa yksinkertaisesti muistiin kirjoitetun tiedon lukemista mahdolliset virheet korjaten ja jopa samaan paikkaan tallentamista [45]. Jos taas käytetään valmista muistiohjainta, sen algoritmeista ei välttämättä ole tarkkaa tietoa, joten virkistys tarkoittaa tiedon lukemista ja uudelleenkirjoittamista samaan loogiseen, mutta kenties eri fyysiseen, osoitteeseen muistilla. Järjestelmän kannalta molemmat vaihtoehdot ovat hyvin lähellä toisiaan, koska molemmissa tilanteissa muistiohjain huolehtii tiedon virheenkorjauksesta ja tallentaa sen samaan loogiseen osoitteeseen, josta se luettiin. Monitasoiset NANDit ovat luotettavuusmielessä huonompia, mutta niiden kapasiteetti on niin suuri nykyään käytettyihin NOReihin verrattuna, että kun kirjoitukset hajautetaan kulutuksentasauksella, solujen kirjoitus- ja tyhjennyskertojen lukumäärien keskiarvo pysyy erittäin matalana ja täten luotettavuus on korkea. Muistityypin vaihto aiheuttaa suuria muutoksia erityisesti ohjelmistoon, mutta muutoksia tarvitaan myös laitteistoon, jos ohjelmakoodi suoritetaan nyt suoraan NOR-piiriltä. Ohjelmistomuutoksia voidaan helpottaa käyttämällä NAND-muisteja, jotka sisältävät jo muistiohjaimen, kuten esimerkiksi muistikortteja. Silti jos vaihdetaan NOR-muistista NANDiin, ainakin muistia käyttävä ajuri joudutaan kirjoittamaan uudestaan. Muistikortti tietovarastona on siinäkin mielessä hyvä konsepti, että niitä käytettäessä latausaseman fyysinen koko voi olla murto-

osa tässä työssä suunnitellusta ja silti se voi tukea moninkertaista määrää yhtäaikaisesti ohjelmoitavia muistipiirejä. NAND-tyyppiset Flash-muistit ovat siis varteenotettavia vaihtoehtoja myös teollisuudessa, mutta muistityypin vaihto on työmäärältään suuri operaatio.

8.2 Ohjelmiston latausasema

Työn käytännön osuutena suunniteltiin toimeksiantaja ABB:lle latausasema, joka syöttää ohjelmiston useaan taajuusmuuttajan ohjauskorttiin yhtäaikaisesti. Laitetta varten suunniteltiin tarvittava elektroniikkaa ja ohjelmistot. Myös mekaniikkakonseptille annettiin suuntaviivoja, rajoja ja vaatimuksia, mutta sen valmistavalle alihankkijalle jätettiin vapaus lopullisen mekaniikan yksityiskohtien suhteen. Latausaseman tavoite on tehostaa toimeksiantajan uuden taajuusmuuttajatuoteperheen tuotantolinjaa.

Valittu konsepti osoittautui hyväksi valinnaksi. Latausasema olisi voinut kirjoittaa taajuusmuuttajan ohjelmiston muistipiirille suoraan, jolloin lataus olisi ollut nopeampaa ja suora hyöty uudesta latausasemasta olisi ollut suurempi. Sen sijaan valittiin ratkaisu, joka käyttää mahdollisimman paljon olemassa olevia ohjelmisto- ja laitteistokomponentteja. Ratkaisu on aavistuksen optimia hitaampi, mutta se ei aiheuta millekään ABB:n tiimille lisätöitä. Näin ollen suunniteltu latausasema on riittävän nopea ja voidaan olettaa sen ylläpitokustannuksien olevan vaihtoehtoista toteutustapaa matalampia. Jatkossa on oletettavaa, että pieniä muutoksia halutaan tehdä asemaan, esimerkiksi mekaniikkaa pitää teettää, kun otetaan mukaan uusia ohjauskortteja, mutta aseman modulaarisuuden vuoksi muutokset ovat pieniä. Jälkikäteen voidaan myös todeta, että teknisesti monimutkaisempi ja työläämpi latausasema, joka suoraan muistipiirille kirjoittava malli on, ei olisi mahtunut diplomityön aikaraameihin.

Suunniteltu laitteen elektroniikka toimii, mutta se olisi voitu toteuttaa kustannustehokkaammin. Lopullinen mekaniikkarakenne poikkesi hieman ensin ajatellusta, joten käytännössä elektroniikka olisi voitu toteuttaa yhdellä piirilevyllä. Kaksi piirilevyä toimii yhtä hyvin ja yhtäältä on modulaarinen ratkaisu, mutta toisaalta ladottujen piirikorttien pienessä tuotantoerässä, jossa kertaluontoisen työn kustannukset ovat valtaosa kokonaisuudesta, kahden piirilevyn ratkaisu lähes kaksinkertaisti valmistuskulut. Ongelma oltaisiin voitu välttää tiiviimmällä yhteistyöllä mekaniikan valmistavan alihankkijan kanssa: alihankkijan kanssa alettiin keskustelemaan mekaniikan valmistuksesta ja sen raameista jo työn alkaessa, mutta yksityiskohtaisemmin mekaniikka suunniteltiin vasta, kun elektroniikka oli jo tilattu.

Projektiin liittyvät ohjelmistot on suunniteltu virhesietoisiksi ja helppokäyttöisiksi. Virhesietoisuutta ovat mikrokontrollerin vahtikoiraominaisuuden käyttö, käyttöliittymän sekä mikrokontrollerin välisessä kommunikoinnissa varautuminen häiriöihin, ja käyttöliittymässä käyttäjän lukemien revisioiden koneellinen varmistus sekä ratkaisuun ohjeistavat virheilmoitukset, joiden avulla virheiltä voidaan välttyä jatkossa. Helppokäyttöisyyteen pyrittiin huomioimalla Nielsenin [62, s. 19–26] käytettävyyden määritelmä ja siihen ohjaava heuristiikka. Tärkeimmät käytettävyyttä edistävät käyttöliittymän suunnitteluratkaisut on listattu luvussa 6.5. Virhesietoisuutta tai käytettävyyttä ei voitu todeta tämän työn puitteissa, koska suunniteltu latausasema ei ole vielä käytössä, joten oikean käyttötilanteen havainnointi oli mahdotonta.

Latausasemaa varten suunniteltu elektroniikka ja ohjelmisto testattiin toimiviksi korjauksin. Pääelektroniikkapiirilevyllä pitää vaihtaa kolme N-tyypin kanavatransistoria toisiin, koska yksi vaihtoehtoisista komponenteista, joka ladottiin piirilevylle, ei ollut yhteensopiva käytetyn hilajännitteen kanssa. Muutos voidaan tehdä siististi käsin juottamalla, joten ei ole tarpeen valmistaa uutta erää piirilevyjä. Suunnitteluvaiheessa ei oltu huomattu, että suuremman tuetun ohjauskortin revisiosignaali ei toimi, kun käytetään vain 5 V käyttöjännitettä. Revisio voidaan lukea kuitenkin viivakoodista, joten tämä ei ole suuri ongelma. Ohjelmistovirheet on korjattu ensimmäisessä virallisesti julkaistussa versiossa, joten ne eivät vaadi jälkitoimia.

Latausasemakokonaisuutta testattaessa todettiin, että kahdesta tuettavasta ohjauskortista pienemmän ohjelmointi onnistui suunnitellusti, mutta suurempi epäonnistui suurella todennäköisyydellä. Epäonnistumisen syy on sen nykyisen latausohjelman versio, joka on työn toimeksiantajan ohjelmistotiimin vastuualueella. On oletettavaa, että laukseen käytetyn ohjelmistoversion päivityksen myötä myös suurempi ohjauskortti voidaan ohjelmoida käytännössä aina onnistuneesti. Tätä ei kuitenkaan voida varmistaa tämän työn puitteissa, koska päivityksen aikataulu on tuntematon.

Työn ensisijainen tutkimuskysymys oli, voidaanko suunnitellulla latausasemalla nopeuttaa taajuusmuuttajien massavalmistusta, ja vastaus on kyllä. Latausasema kuusinkertaistaa yhden operaattorin työtehon tehtaan lattiapinta-alaa kohden aseman edeltäjään verrattuna. Varsinainen työtehon kasvu riippuu siitä, kuinka montaa latausasemaa yksi operaattori tulee käyttämään yhtä aikaa. Yksi latausasema ei välttämättä riitä kattamaan kaikkia ABB:n uuden taajuusmuuttajatuoteperheen ohjelmointitarpeita, mutta vaikka latausasemia valmistettaisiin useampia, työntekijöitä ei tarvita niin montaa tuotantolinjalla kuin mitä heitä olisi tarvittu ilman tässä työssä suunniteltua latausasemaa.

Työhön liittyy muutamia mahdollisia jatkokehitysmahdollisuuksia. Varma jatkokehitys on testata suuremman ohjauskortin ohjelmointia, kun komentorivipohjaisen latausohjelman varmemmin toimiva versio tulee saataville. Mikäli latausasema toimii tuotannossakin odotetusti, myös muita ohjauskortteja voidaan ohjelmoida latausasemalla. Tämä vaatii ainoastaan sopivan neulapedin teettämistä. ABB:lla on kehitteillä lisäksi mekaanisesti huomattavan pieniä ohjauskortteja, joita saatetaan haluta ohjelmoida kuuden sijaan jopa 12 kappaletta kerrallaan. Tällöin laitteistoa pitää monistaa, mutta mitään uutta ei tarvitse suunnitella, ja käyttöliittymään tulee tehdä pieniä muutoksia. Työn aikana esiin nousi myös huomio, että tietyissä tilanteissa ohjelmisto halutaan päivittää myös valmiisiin, koottuihin taajuusmuuttajiin. Latausasemaa varten suunniteltua käyttöliittymää voidaan käyttää tässä tilanteessa helpottamaan operaattorin työtä ilman latausaseman laitteistoa.

Viitteet

[1] Euroopan unioni, Neuvosto. Direktiivi 73/23/ETY (tietyllä jännitealueella toimivia sähkölaitteita koskevan jäsenvaltioiden lainsäädännön lähentämisestä). Direktiivi. 19.2.1973. [Viitattu 19.2.2013]. URL http://eur-lex.europa.eu/LexUriServ/LexUriServ.do?uri=CELEX:31973L0023:FI:NOT.

[2] ABB. ABB general purpose drive, ACS550-sarjan taajuusmuuttajat. Tuoteluettelo. 2013. [Viitattu 5.8.2013]. URL http://www05.abb.com/global/scot/scot201.nsf/veritydisplay/fdaf7a45127b1285c1257ba400299c07/$file/FI_ACS550catalogREVP_July2013.pdf.

[3] Niiranen, J. Sähkömoottorikäytön digitaalinen ohjaus. Helsinki: Otatieto 2000. 381 s. ISBN 951-672-300-4.

[4] Atmel. Atmel AVR231: AES Bootloader, revisio 2589E-AVR-03/12. Sovellusohje. 2012.

[5] Skorobogatov, S. Semi-invasive attacks – A new approach to hardware security analysis. 4.2005. ISSN 1476-2986.

[6] JEDEC. JESD79-4, DDR4 SDRAM. Standardi. 9.2012. [Viitattu 18.2.2013]. URL www.jedec.org/sites/default/files/docs/JESD79-4.pdf.

[7] Bez, R. & Camerlenghi, E. & Modelli, A. & Visconti, A. Introduction to flash memory. Proceedings of the IEEE 2003. Vol. 91:4. S. 489–502. DOI:10.1109/JPROC.2003.811702. ISSN 0018-9219.

[8] Micheloni, R. & Crippa, L. & Marelli A. Inside NAND Flash Memories. Springer 2010. 592 s. ISBN 978-9048194308.

[9] Brewer, R. & Gill, M. Nonvolatile Memory Technologies with Emphasis on Flash. New Jersey, USA: John Wiley & Sons 2008. 792 s. ISBN 978-0471770022.

[10] Campardo, G. & Micheloni, R. & Novosel, D. Inside NAND Flash Memories. Berlin, Saksa: Springer 2005. 740 s. ISBN 978-3540201984.

[11] Aritome, S. & Shirota, R. & Hemink, G. & Endoh, T. & Masuoka, F. Reliability issues of flash memory cells. Proceedings of the IEEE 1993. Vol. 81:5. S. 776–788. DOI:10.1109/5.220908. ISSN 0018-9219.

[12] Sedra, A. & Smith, K. Microelectronic Circuits. New York, USA: Oxford University Press 2004. 1392 s. ISBN 978-0195338836.

[13] Suh K.-D. & Suh B.-H. & Lim Y.-H. & Kim J.-K. & Choi Y.-J. & Koh Y.-N. & Lee S.-S. & Kwon S.-C. & Choi B.-S. & Yum J.-S. & Choi J.-H. & Kim J.-R. & Lim H.-K. A 3.3 V 32 Mb NAND flash memory with incremental step pulse programming scheme. Solid-State Circuits, IEEE Journal of 1995. Vol. 30:11. S. 1149–1156. DOI:10.1109/4.475701. ISSN 0018-9200.

[14] Van Houdt, J. Flash memory: a challenged memory technology. Teoksessa: Integrated Circuit Design and Technology, 2006. ICICDT '06. 2006 IEEE International Conference, 2006. S. 1–4. (DOI:10.1109/ICICDT.2006.220787).

[15] Lu, C.-Y. & Lu, T.-C. & Liu, R. Non-Volatile Memory Technology – Today and Tomorrow. Teoksessa: Physical and Failure Analysis of Integrated Circuits, 2006. 13th International Symposium on the 2006. S. 18–23. (DOI:10.1109/IPFA.2006.250989).

[16] Bez, R. & Cappelletti, P. Flash memory and beyond. Teoksessa: VLSI Technology, 2005. (VLSI-TSA-Tech). 2005 IEEE VLSI-TSA International Symposium, 2005. S. 84–87. (DOI:10.1109/VTSA.2005.1497090).

[17] Lin, J.-H. & Chang, Y.-H. & Hsieh, J.-W. & Kuo, T.-W. & Yang, C.-C. A NOR Emulation Strategy over NAND Flash Memory. Teoksessa: Embedded and Real-Time Computing Systems and Applications, 2007. RTCSA 2007. 13th IEEE International Conference on 2007. S. 95–102. ISSN 1533-2306 (DOI:10.1109/RTCSA.2007.9).

[18] Shin, Y Non-volatile memory technologies for beyond 2010. Teoksessa: VLSI Circuits, 2005. Digest of Technical Papers. 2005 Symposium on 2005. S. 156–159. (DOI:10.1109/VLSIC.2005.1469355).

[19] Mikolajick, T. & Specht, M. & Nagel, N. & Mueller, T. & Riedel, S. & Beug, F. & Melde, T. & Kusters, K.-H. The Future of Charge Trapping Memories. Teoksessa: VLSI Technology, Systems and Applications, 2007. VLSI-TSA 2007. International Symposium on 2007. S. 1–4. ISSN 1524-766X (DOI:10.1109/VTSA.2007.378943).

[20] Pavan, P. & Bez, R. & Olivo, P. & Zanoni, E. Flash memory cells – An overview. Proceedings of the IEEE 1997. Vol. 85:8. S. 1248–1271. DOI:10.1109/5.622505. ISSN 0018-9219.

[21] Micheloni, R. & Picca, M. & Amato, S. & Schwalm, H. & Scheppler, M. & Commodaro, S. Non-Volatile Memories for Removable Media. Proceedings of the IEEE 2009. Vol. 97:1. S. 148–160. DOI:10.1109/JPROC.2008.2007477. ISSN 0018-9219.

[22] Chien, R. Spansion Overtakes Micron in NOR Revenue for First Time Since 2009. Lehdistötiedote. 4.9.2012. [Viitattu 17.6.2013]. URL http://www.isuppli.com/Memory-and-Storage/MarketWatch/Pages/Spansion-Overtakes-Micron-in-NOR-Revenue-for-First-Time-Since-2009.aspx.

[23] Chien, R. Surprise! NAND Flash Market Defies Trends and Grows to Record Level in Q4. Lehdistötiedote. 26.3.2013. [Viitattu 17.6.2013]. URL http://www.isuppli.com/Memory-and-Storage/News/Pages/Surprise-NAND-Flash-Market-Defies-Trends-and-Grows-to-Record-Level-in-Q4.aspx.

[24] Eon Silicon Solution. EN25QH64, revisio H. Datalehti. 5.11.2012.

[25] Macronix International. MX25L6435E, revisio 1.1. Datalehti. 25.2.2013.

[26] Micron. M25PX64, revisio B. Datalehti. 3/2013.

[27] Micron. N25Q064A, revisio J. Datalehti. 1/2013.

[28] Spansion. S25FL064P, revisio 08. Datalehti. 29.1.2013.

[29] Winbond. S25FL064P, revisio F. Datalehti. 15.10.2012.

[30] Adesto Technologies. AT45DB642D, revisio M. Datalehti. 11/2012.

[31] Eon Silicon Solution. EN29GL064, revisio N. Datalehti. 28.12.2011.

[32] Macronix International. MX29LV640E T/B, revisio 1.7. Datalehti. 27.12.2011.

[33] Micron. PC28F064M29EWXX, revisio B. Datalehti. 11/2012.

[34] Spansion. S29GL064N, revisio 12. Datalehti. 29.10.2008.

[35] Winbond. W29GL064C, revisio F. Datalehti. 3.8.2012.

[36] Eon Silicon Solution. EN27LN1G08, revisio B. Datalehti. 30.12.2011.

[37] Numonyx (nykyään Micron). NAND128-A, revisio 17. Datalehti. 19.10.2012.

[38] Spansion. S34ML01G1, revisio 08. Datalehti. 2.8.2012.

[39] Toshiba. TC58DVG02D5TA00, revisio 1.10. Datalehti. 31.8.2011.

[40] JEDEC. JESD68.01, Common Flash Interface (CFI). Standardi. 9.2003. [Viitattu 20.6.2013]. URL http://www.jedec.org/sites/default/files/docs/jesd68-01.pdf.

[41] JEDEC. JESD216, Serial Flash Discoverable Parameters (SFDP), for Serial NOR Flash. Standardi. 4.2011. [Viitattu 18.6.2013]. URL http://www.jedec.org/sites/default/files/docs/JESD216.pdf.

[42] Tseng, H.-W. & Grupp, L. & Swanson, S. Understanding the impact of power loss on flash memory. Teoksessa: Design Automation Conference (DAC), 2011 48th ACM/EDAC/IEEE 2011. S. 35–40. ISSN 0738-100x.

[43] Venkat, K. & Haensel, U. Understanding MSP430 Flash Data Retention (SLAA392). Sovellusohje. 3/2008.

[44] Forstner, P. MSP430 Flash Memory Characteristics (SLAA334A). Sovellusohje. 4/2008.

[45] Cai Y. & Yalcin, G. & Mutlu, O. & Haratsch, E. & Cristal, A. & Unsal, O. & Mai, K. Flash correct-and-refresh: Retention-aware error management for increased flash memory lifetime. Teoksessa: Computer Design (ICCD), 2012 IEEE 30th International Conference on, 2012. S. 94–101. ISSN 1063-6404 (DOI:10.1109/ICCD.2012.6378623).

[46] Brand, A. & Wu, K. & Pan, S. & Chin, D. Novel read disturb failure mechanism induced by FLASH cycling. Teoksessa: Reliability Physics Symposium, 1993. 31st Annual Proceedings., International 1993. S. 127–132. (DOI:10.1109/RELPHY.1993.283291).

[47] Moon, T. Error Correction Coding: Mathematical Methods and Algorithms. New Jersey, USA: John Wiley & Sons 2005. 800 s. ISBN 978-0-471-64800-0.

[48] Stievano, I. Flash Memories. Croatia: InTech 2011. 262 s. ISBN 978-953-307-272-2.

[49] Sun, H. & Grayson, P. & Wood, B. Quantifying reliability of solid-state storage from multiple aspects. Teoksessa: Proc. of the 7th IEEE International Workshop on Storage Network Architecture and Parallel I/Os (SNAPI) 2011 .

[50] Yaakobi, E. & Grupp, L. & Siegel, P. & Swanson, S. & Wolf, J. Characterization and error-correcting codes for TLC flash memories. Teoksessa: Computing, Networking and Communications (ICNC), 2012 International Conference on 2012. S. 486–491. (DOI:10.1109/ICCNC.2012.6167470).

[51] Cai, Y. & Haratsch, E. & Mutlu, O. & Mai, K. Error patterns in MLC NAND flash memory: Measurement, characterization, and analysis. Teoksessa: Design, Automation Test in Europe Conference Exhibition (DATE), 2012 2012. S. 521–526. ISSN 1530-1591 (DOI:10.1109/DATE.2012.6176524).

[52] Wong, W. The Path to High-Performance NAND Flash. Electronic Design. 7.3.2013.

[53] JEDEC. JESD230, NAND Flash Interface Interoperability. Standardi. 10.2012. [Viitattu 29.6.2013]. URL `http://www.jedec.org/sites/default/files/docs/JESD230.pdf`.

[54] Wilson, P. The Circuit Designer's Companion. Oxford, UK: Newnes 2012. 456 s. ISBN 978-0080971384.

[55] International Standard - ISO/IEC 14764 IEEE Std 14764-2006 Software Engineering — Software Life Cycle Processes — Maintenance. ISO/IEC 14764:2006 (E) IEEE Std 14764-2006 Revision of IEEE Std 1219-1998) 2006. S. 1–46. DOI:10.1109/IEEESTD.2006.235774.

[56] Eick, S. & Graves, T. & Karr, A. & Marron, J. & Mockus, A. Does code decay? Assessing the evidence from change management data. Software Engineering, IEEE Transactions on 2001. Vol. 27:1. S. 1–12. DOI:10.1109/32.895984. ISSN 0098-5589.

[57] Compaq & Hewlett-Packard & Intel & Lucent & Microsoft & NEC & Philips. Universal Serial Bus Specification, Revision 2.0. Standardi. 27.4.2000. [Viitattu 22.7.2013]. URL http://www.usb.org/developers/docs.

[58] Morrison, R. Grounding and Shielding: Circuits and Interference, 5. painos. New Jersey, USA: John Wiley & Sons 2007. 208 s. ISBN 978-0470097724.

[59] STMicroelectronics. STM32F3DISCOVERYDiscovery kit for STM32F303xx microcontrollers. Tuotekuvaus. 2013. [Viitattu 3.8.2013]. URL http://www.st.com/web/en/catalog/tools/PF254044.

[60] Tarnoff, D. Computer Organization and Design Fundamentals: Examining Computer Hardware from the Bottom to the Top. David L. Tarnoff & Lulu.com 2007. 408 s.

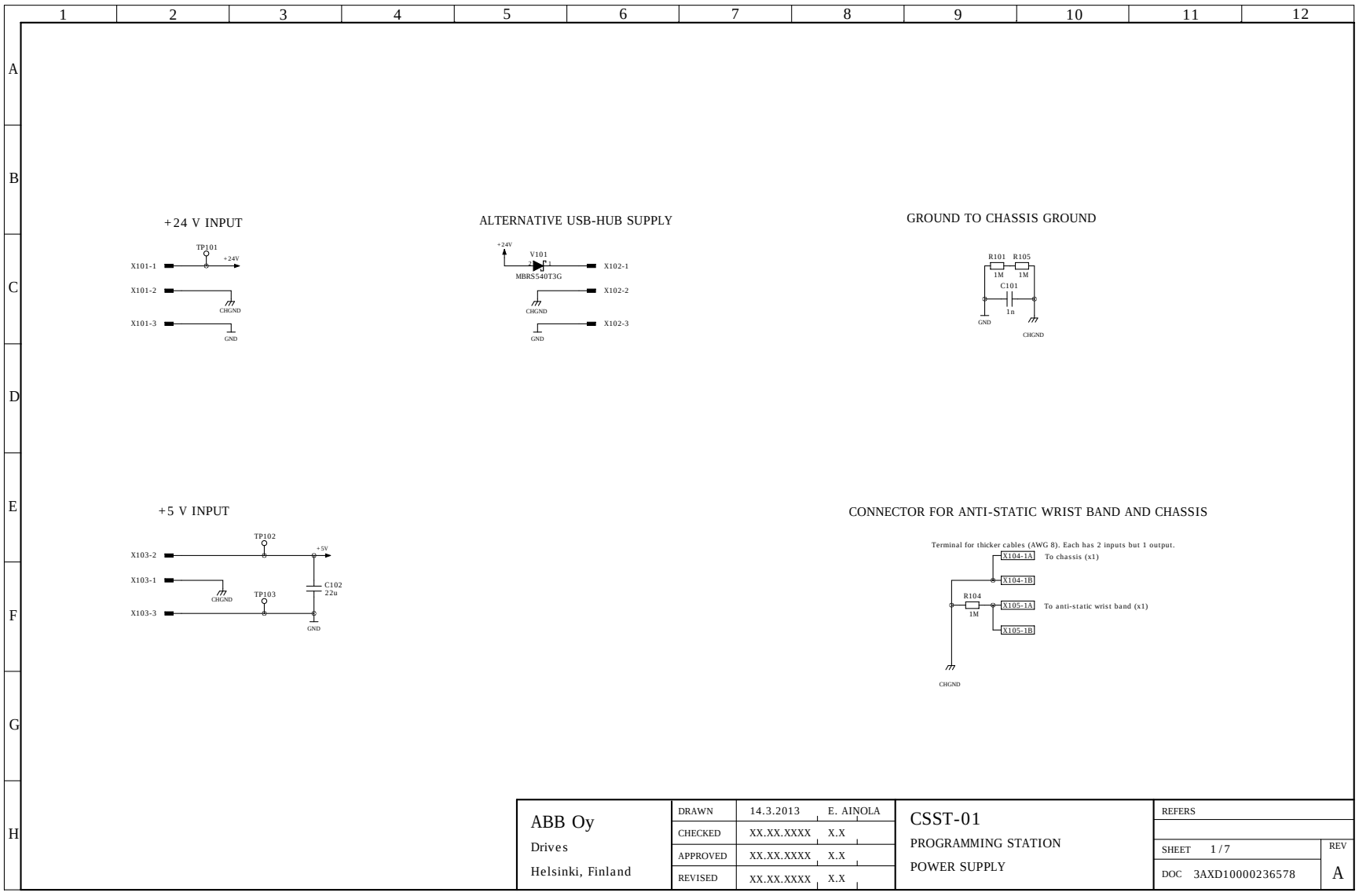
[61] Feldmeier, D. Fast software implementation of error detection codes. Networking, IEEE/ACM Transactions on 1995. Vol. 3:6. S. 640–651. DOI:10.1109/90.477710. ISSN 1063-6692.

[62] Nielsen, J. Usability Engineering. London, UK: Academic Press 1993. 362 s. ISBN 978-0125184069.

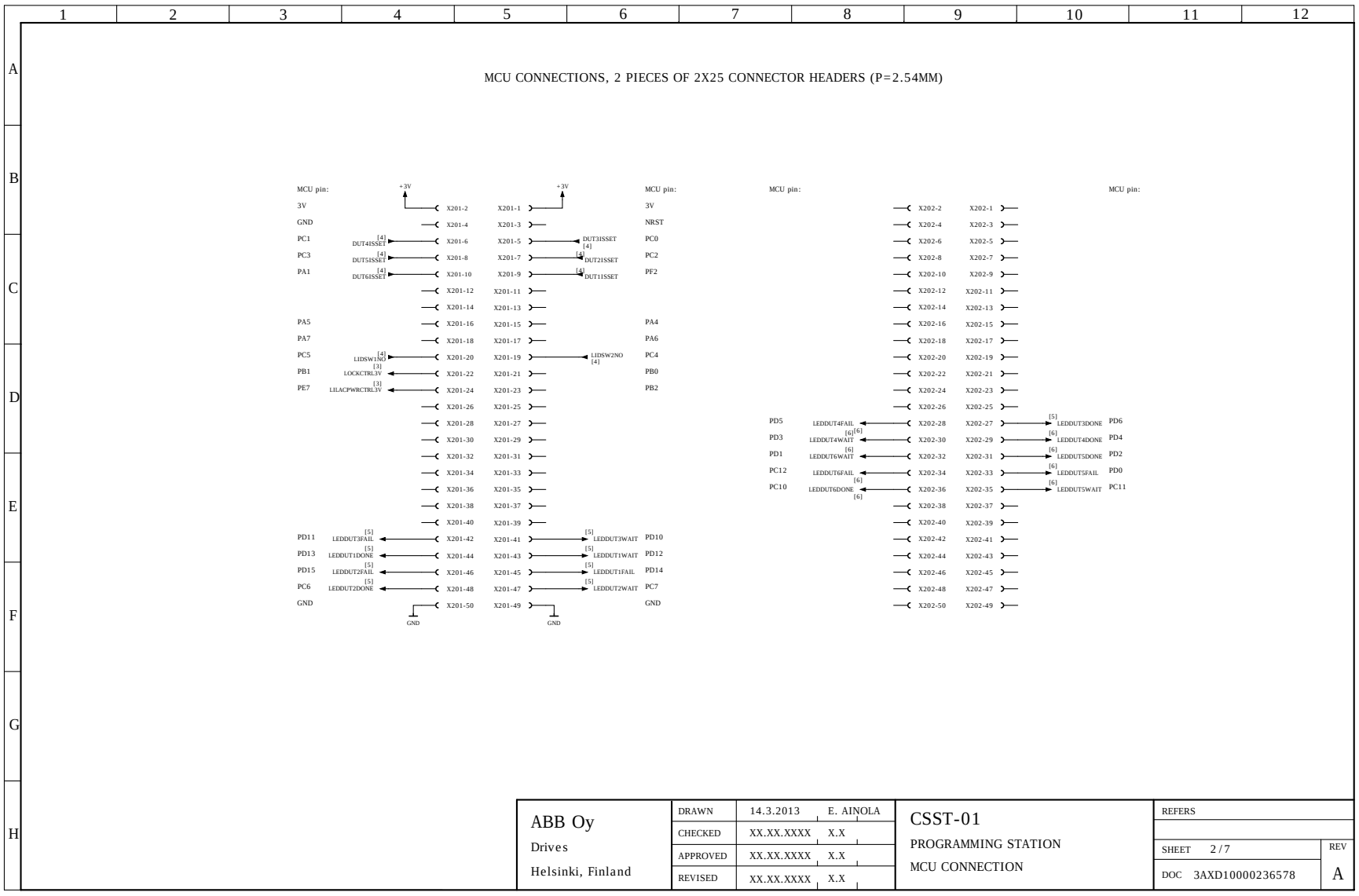
A Liite: Latausaseman elektroniikan piirikaaviot

Tässä liitteessä esitetään pääelektroniikan piirikaavio kuvissa A.1–A.7 ja neulapetielektroniikan vastaava kuvissa A.8–A.14. Osaluetteloa vaihtoehtoisine komponentteineen ei listata, koska se sisältää luottamuksellista tietoa.

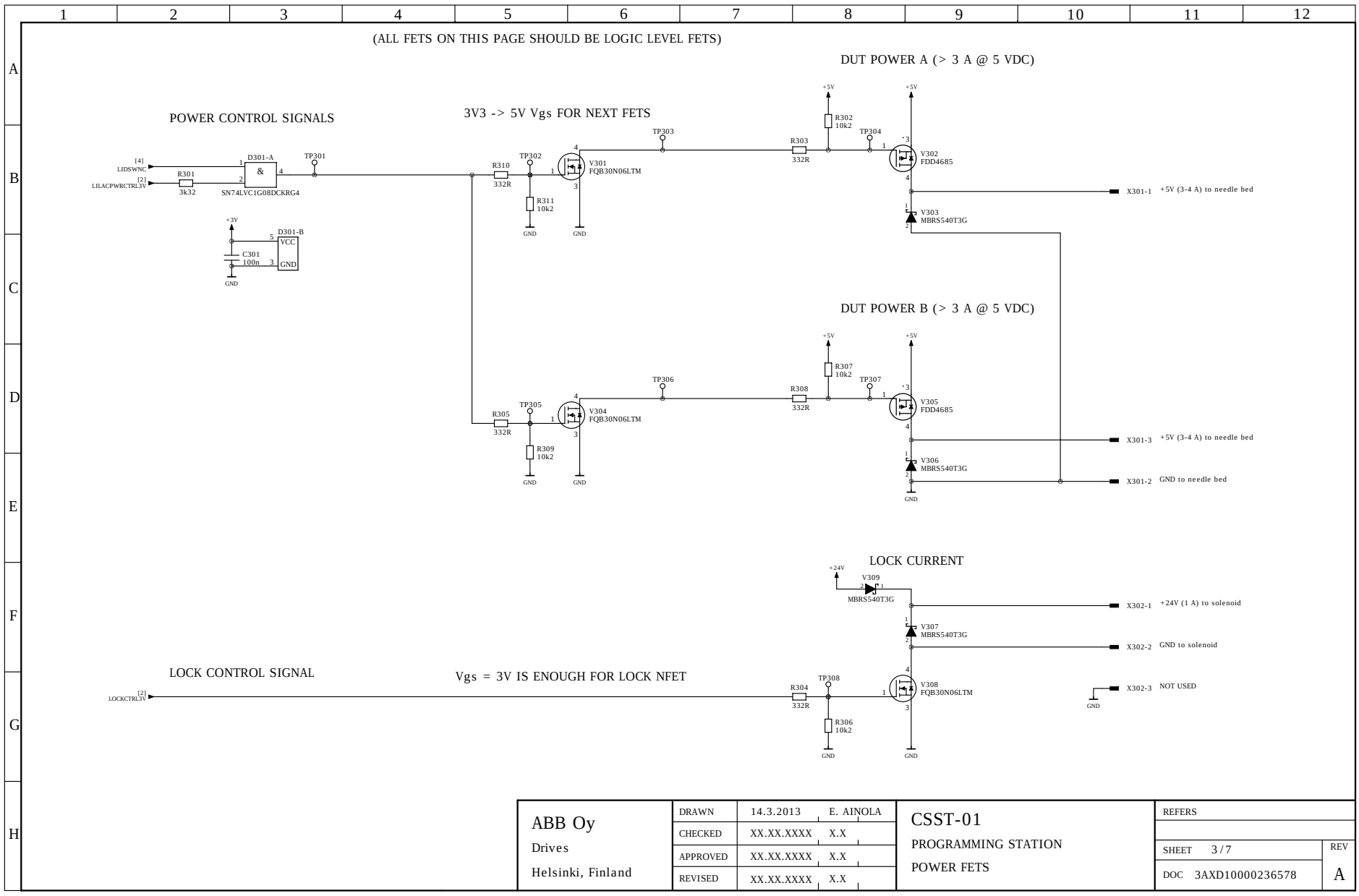
A.1 Pääelektroniikan piirikaavio



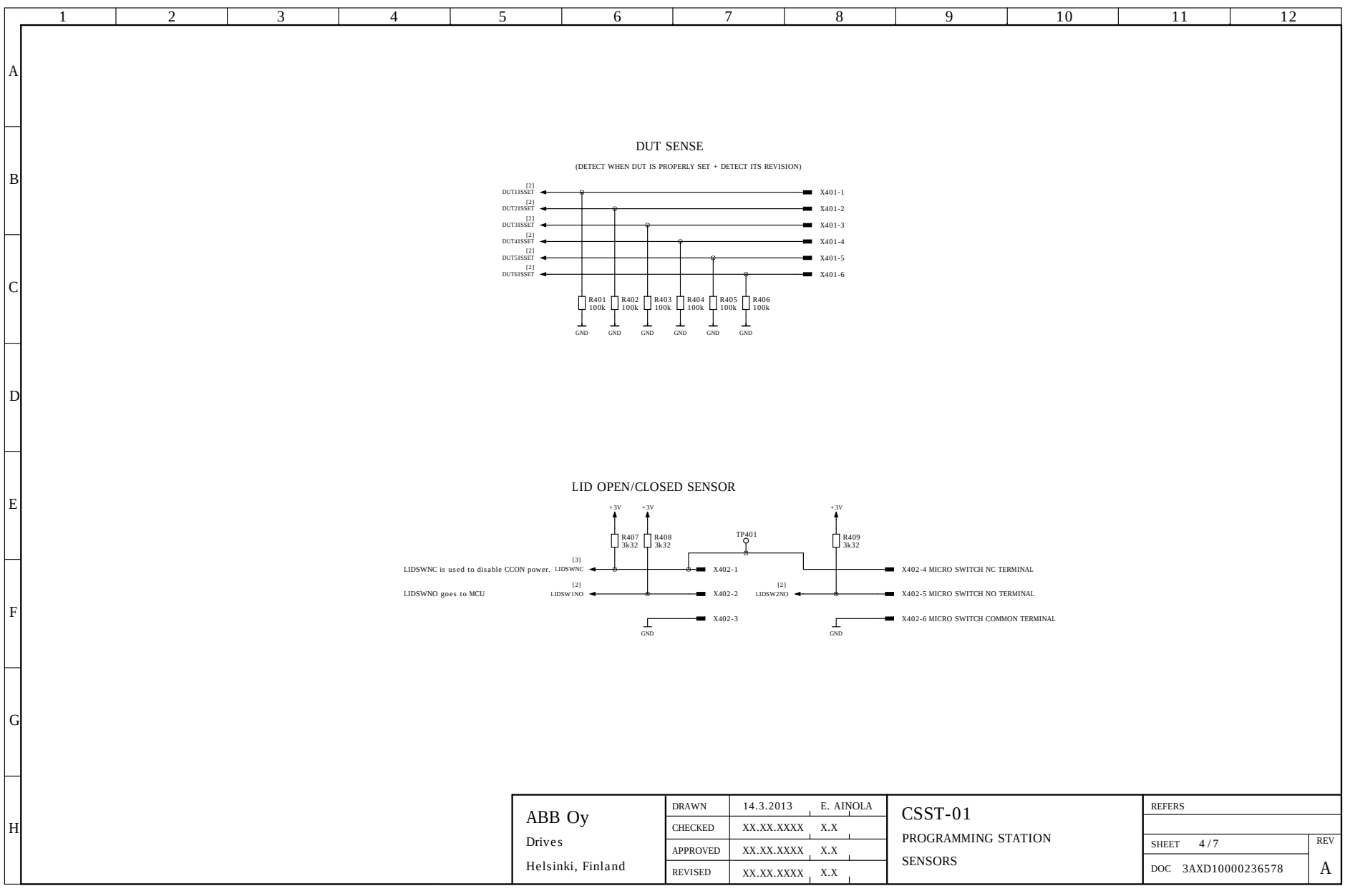
Kuva A.1: Käyttöjännitteen sisääntulo sekä maadoituskytkennät.



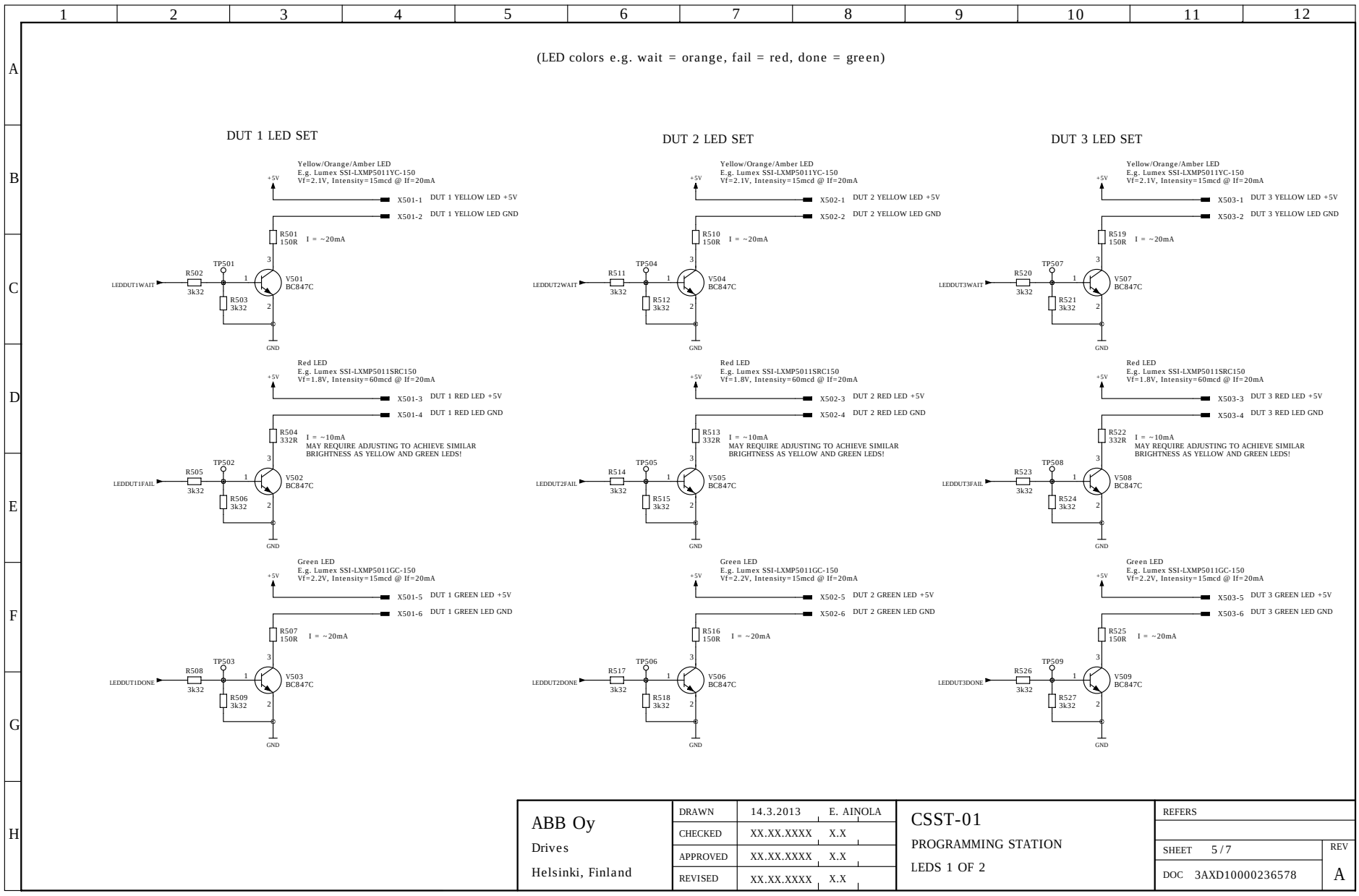
Kuva A.2: Liityntä testialustalla olevaan mikrokontrolleriin.



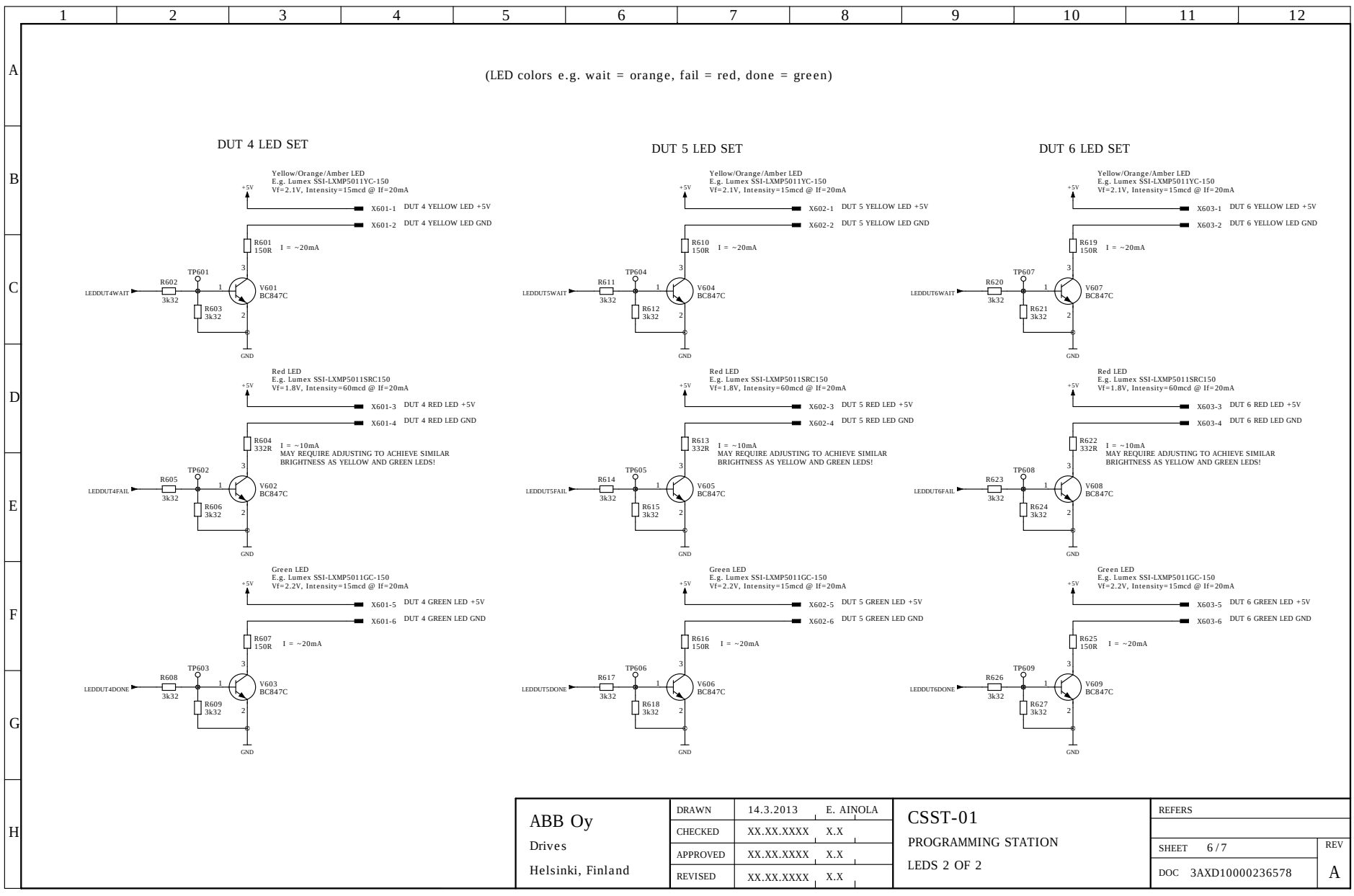
Kuva A.3: Neulapetielektroniikan käyttöjännitteen ja lukon ohjaus.



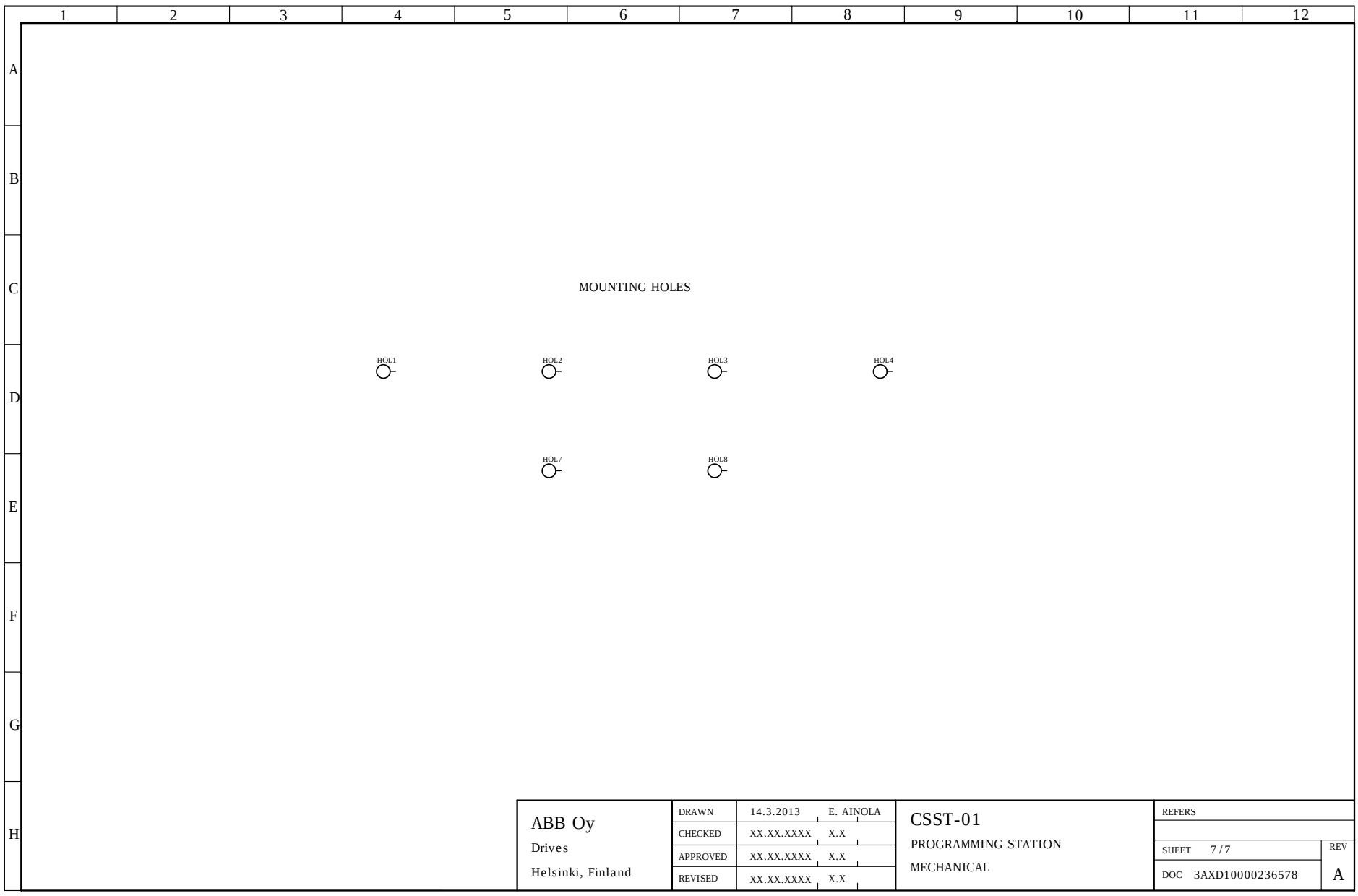
Kuva A.4: Ohjauskorttien revisioiden ja mikrokytkinten tilan lukeminen.



Kuva A.5: Tilan ilmoittavien valodiodien ohjaus kolmelle ensimmäiselle latausaseman ohjelmointipaikalle.

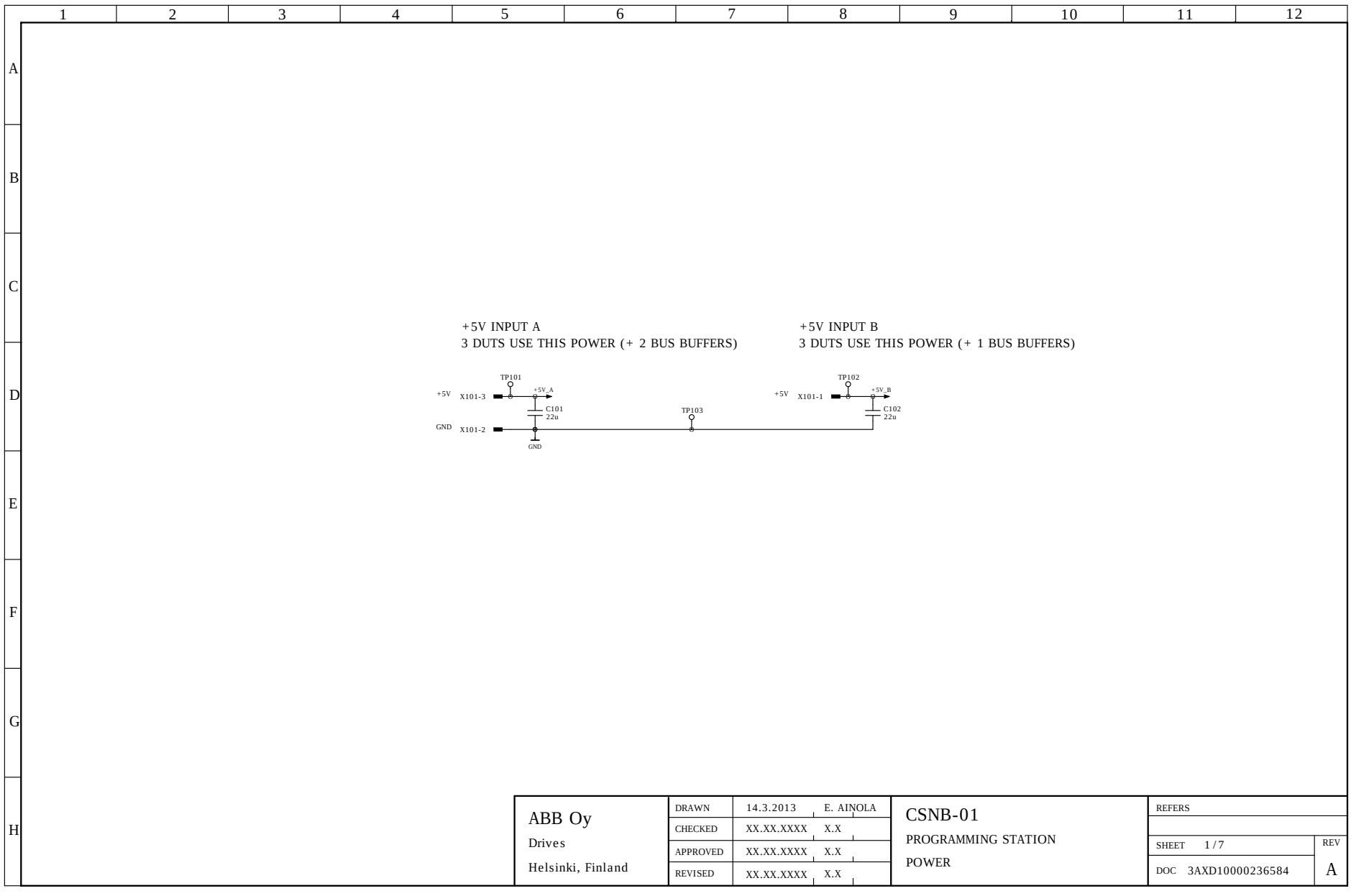


Kuva A.6: Tilan ilmoittavien valodiodien ohjaus kolmelle viimeiselle latausaseman ohjelmointipaikalle.

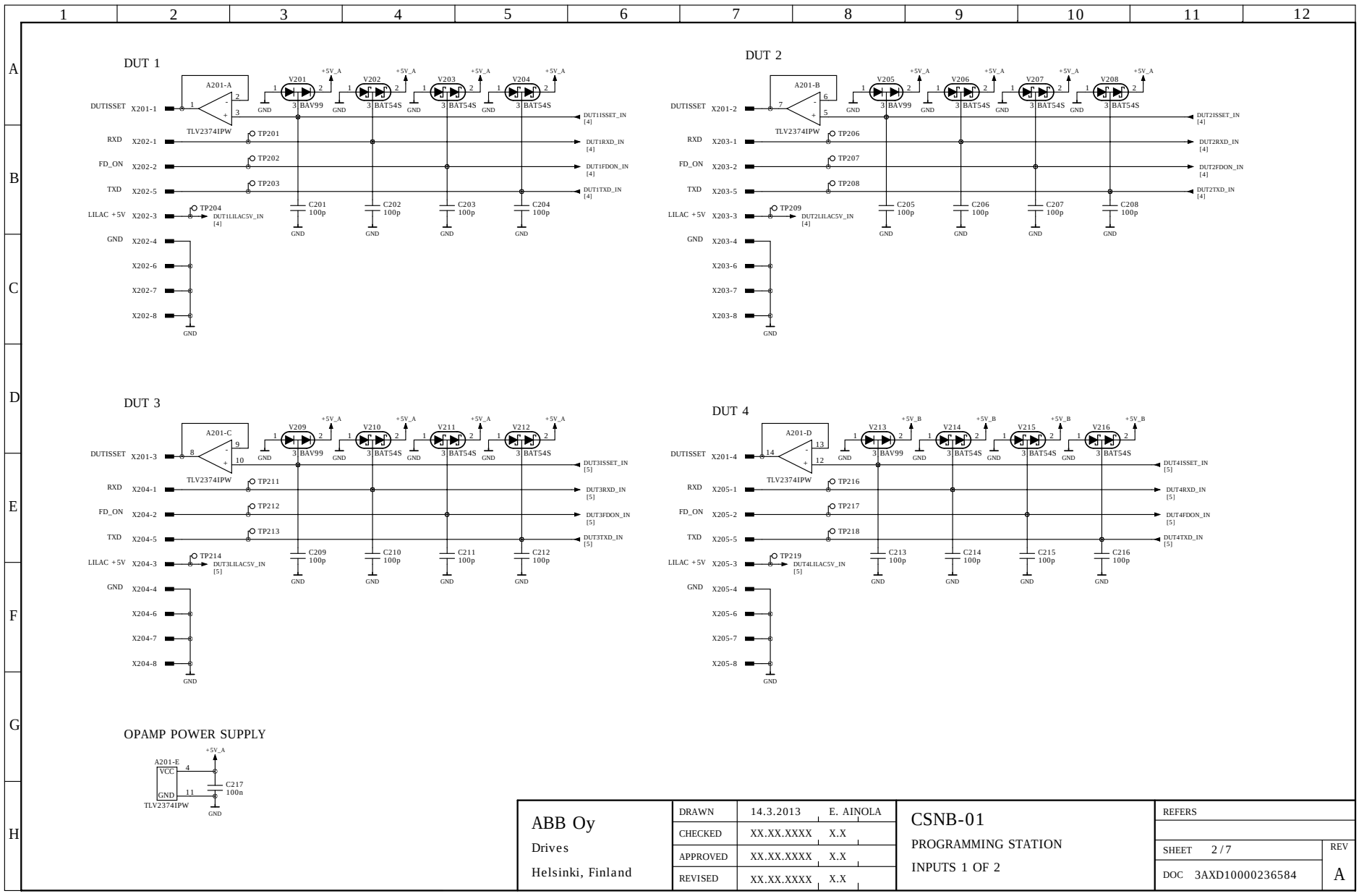


Kuva A.7: Kiinnitysreiät.

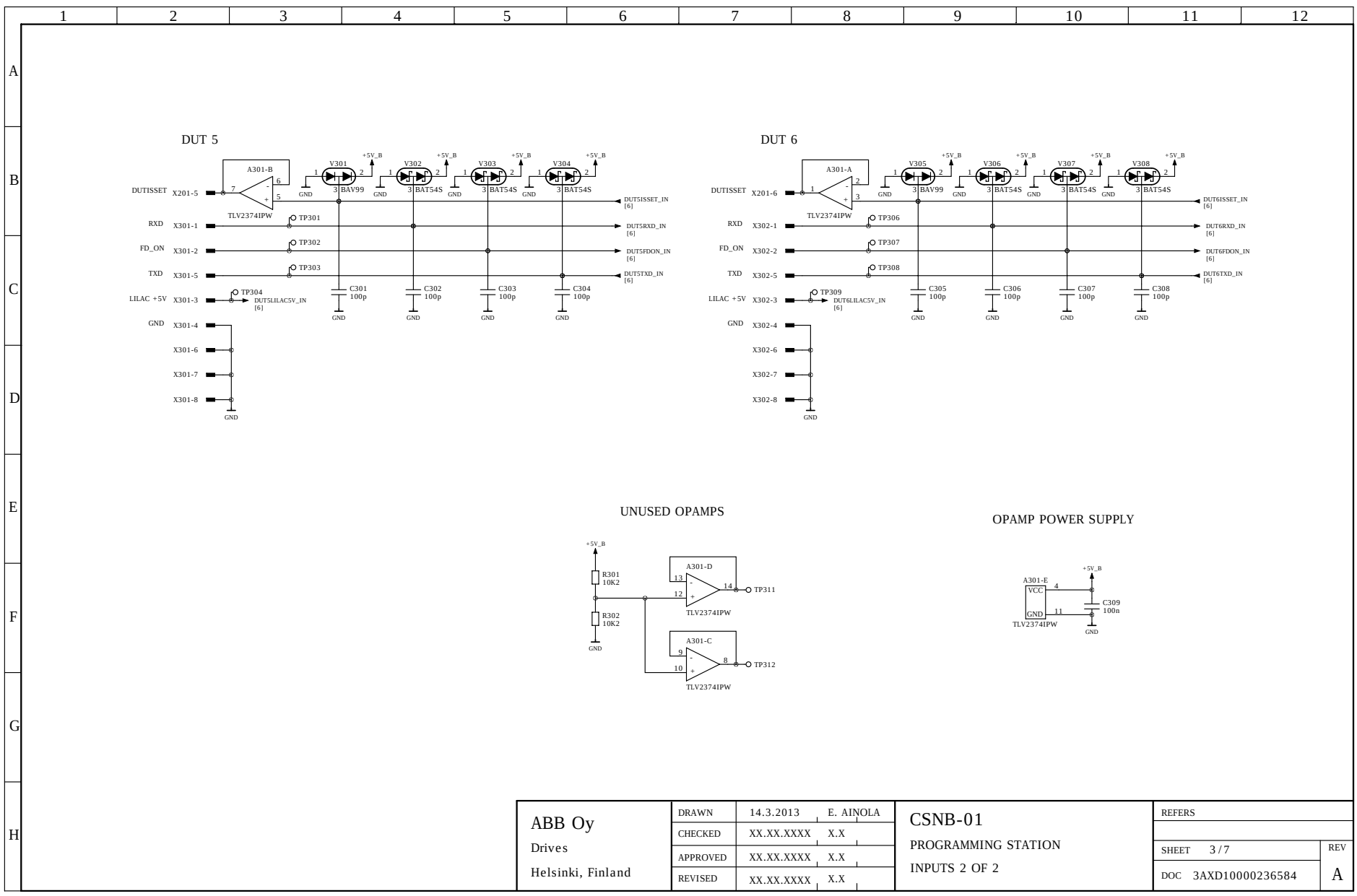
A.2 Neulapetielektroniikan piirikaavio



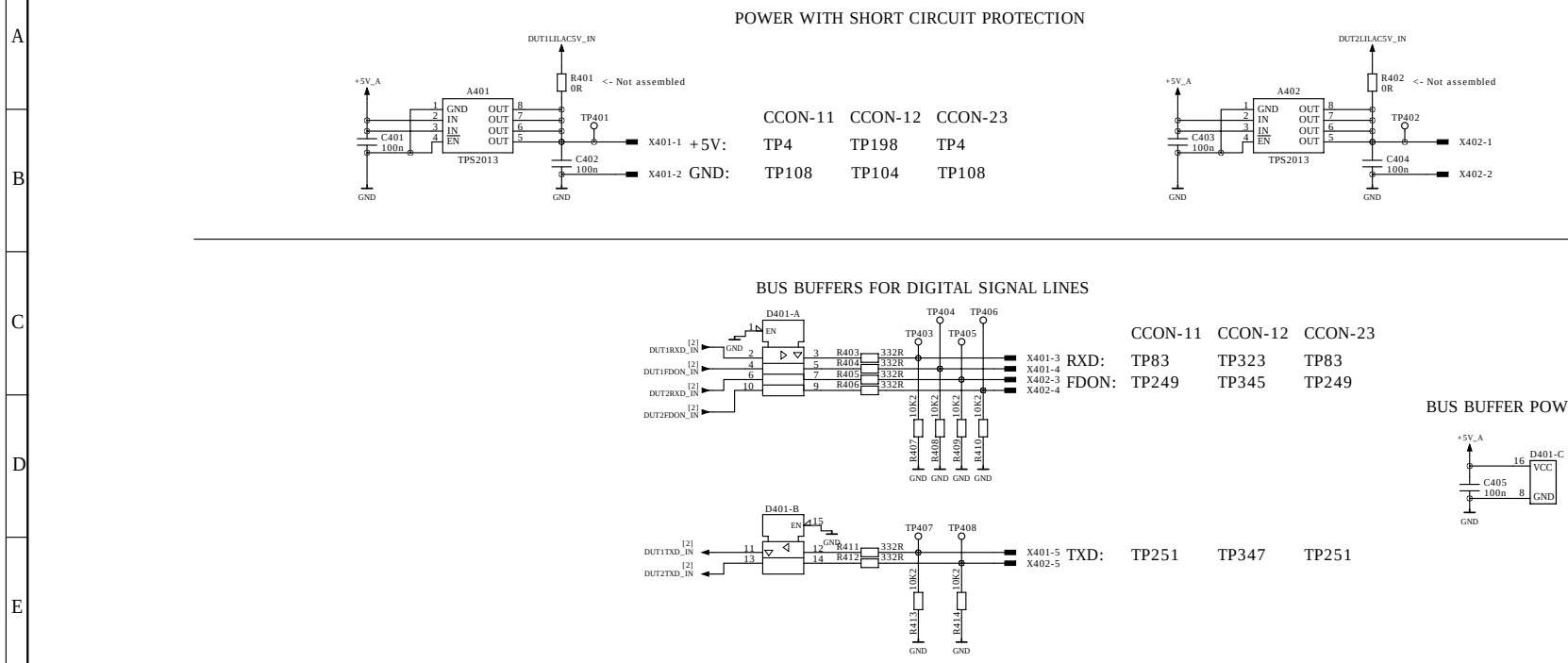
Kuva A.8: Käyttöjännitteen sisääntulo.



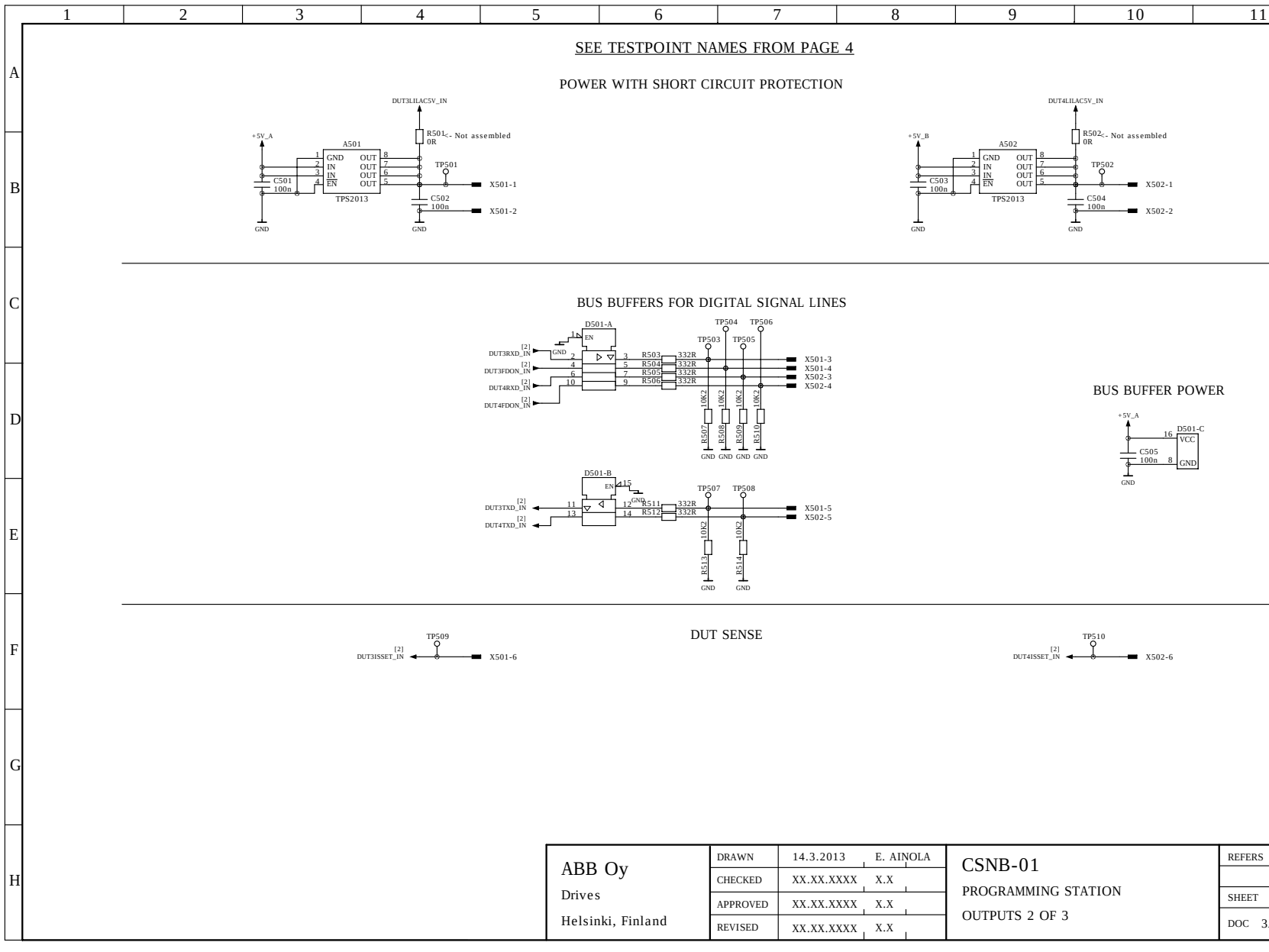
Kuva A.9: Neljän ensimmäisen ohjauskortin analogisen revisiosignaalin puskurointi ja digitaalisten signaalien sisääntulo USB–TIA/EIA-485-muuntimelta.



Kuva A.10: Kahden viimeisen ohjauskortin analogisen revisiosignaalin puskurointi ja digitaalisten signaalien sisääntulo USB–TIA/EIA-485-muuntimelta.



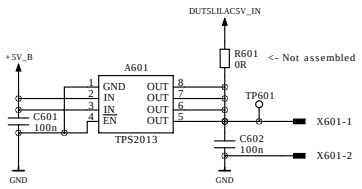
Kuva A.11: Kahden ensimmäisen ohjauskortin oikosulkusuojaus sekä niiden digitaalisten signaalien puskurointi.

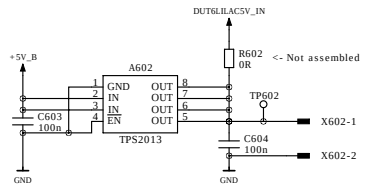


Kuva A.12: Kahden seuraavan ohjauskortin oikosulkusuojaus sekä niiden digitaalisten signaalien puskurointi.

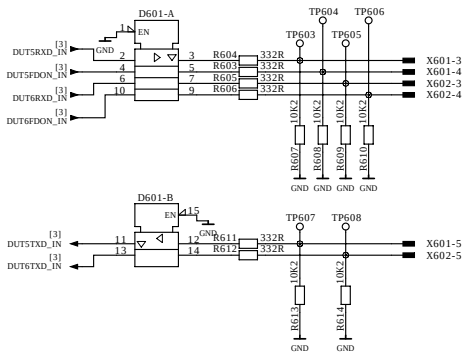
SEE TESTPOINT NAMES FROM PAGE 4

POWER WITH SHORT CIRCUIT PROTECTION

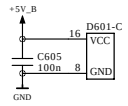




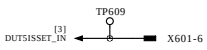
BUS BUFFERS FOR DIGITAL SIGNAL LINES



BUS BUFFER POWER



DUT SENSE



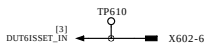


ABB Oy Drives Helsinki, Finland	DRAWN	14.3.2013 E. AINOLA	CSNB-01 PROGRAMMING STATION OUTPUTS 3 OF 3	REFERS	
	CHECKED	XX.XX.XXXX X.X		SHEET 6 / 7	REV
	APPROVED	XX.XX.XXXX X.X		DOC 3AXD10000236584	A
	REVISED	XX.XX.XXXX X.X			

Kuva A.13: Kahden viimeisen ohjauskortin oikosulkusuojaus sekä niiden digitaalisten signaalien puskurointi.

MOUNTING HOLES

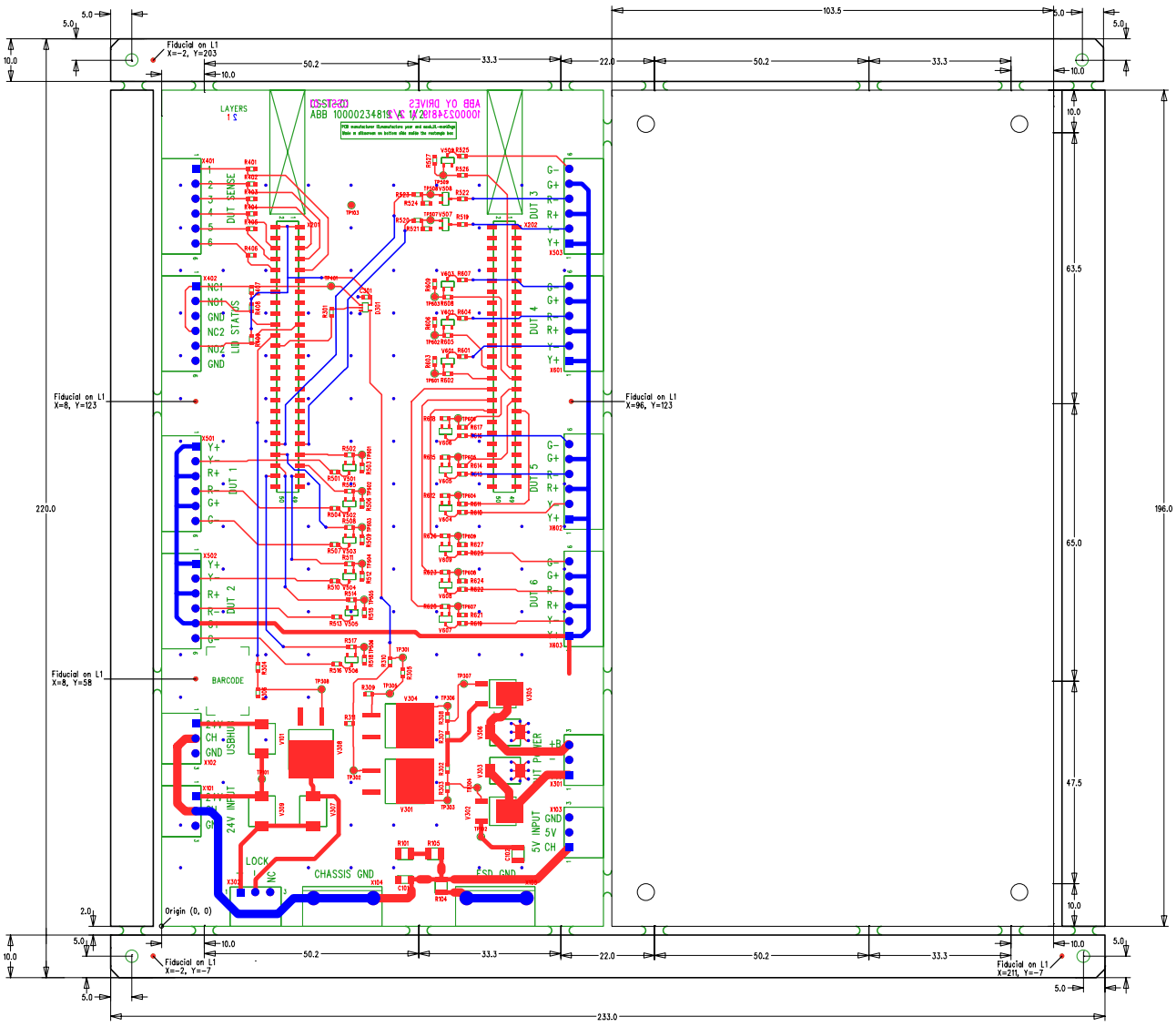
ABB Oy Drives Helsinki, Finland	DRAWN	14.3.2013 E. AINOLA	CSNB-01 PROGRAMMING STATION MECHANICAL	REFERS	
	CHECKED	XX.XX.XXXX X.X		SHEET 7 / 7	REV
	APPROVED	XX.XX.XXXX X.X		DOC 3AXD10000236584	A
	REVISED	XX.XX.XXXX X.X			

Kuva A.14: Kiinnitysreiät.

B Liite: Latausaseman elektroniikan asettelukaaviot

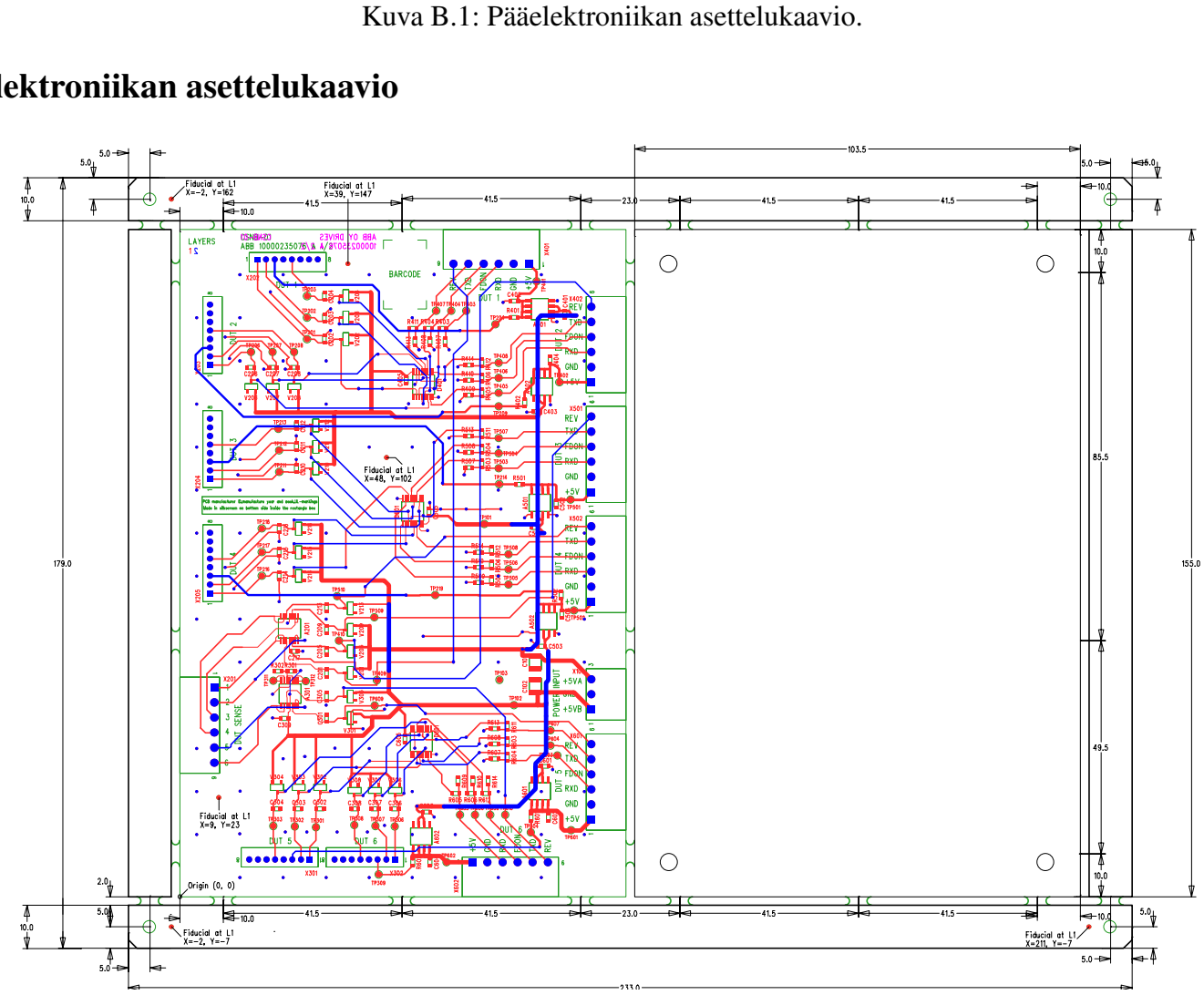
Tässä liitteessä esitetään pääelektroniikan asettelukaavio kuvassa B.1 ja neulapetielektroniikan vastaava kuvassa B.2. Asettelukaavioon kuuluu useita kerroksia eri valmistusvaiheita varten. Seuraavissa kuvissa mukana on vain kuparikerrokset (punainen on yläkerros, sininen on alakerros), pintapainatuskerros (vihreä on yläkerros, vaaleanpunainen on alakerros) ja panelointipiirros (musta). Muut on jätetty pois luettavuuden helpottamiseksi. Näitä ovat muun muassa reiät, komponenttien latomiskaavio, *juotospeite (engl. paste mask)*, *juotoseste (engl. solder mask)* ja suuret kuparialueet, kuten *maataso (engl. ground plane)*. Mitat on ilmoitettu millimetreinä ja koordinaatiston origo on merkitty piirilevyn vasempaan alakulmaan.

B.1 Pääelektroniikan asettelukaavio



Kuva B.1: Pääelektroniikan asettelukaavio.

B.2 Neulapetielektroniikan asettelukaavio



Kuva B.2: Neulapetielektroniikan asettelukaavio.

C Liite: Latausaseman mikrokontrollerin lähdekoodi

Seuraavassa on latausaseman mikrokontrolleriohjelmiston lähdekoodi. Se perustuu STMicroelectronicsin esimerkkiprojektiin `VirtualComport_Loopback`, joka löytyy STMicroelectronicsin Internet-sivuilta paketista `STM32_USB-FS-Device_Lib_V4.0.0`. Se sisältää useita esimerkkejä STMicroelectronicsin mikrokontrollerikokeilualustoille, jotta niiden ominaisuuksia on helpompi kokeilla. Seuraava koodilistaus sisältää vain tätä projektia varten kirjoitetun koodin. Se käyttää edellä mainitun esimerkkipaketin toteuttamia funktioita ja vakioita.

C.1 Mikrokontrolleriohjelmiston otsikkotiedosto

```
/**
  ************************************************
  * @file    main.h
  * @author  Eppu Ainola
  * @date    2013-06-12
  * @brief   Header for Main Program of Software Loading Station
  ************************************************
  * @attention
  *     This program (more specifically the included init files) are from STMicroelectronics' (STM) STM32_USB-FS-Device_Lib_V4.0.0 VirtualComport_Loopback. In addition init procedures from STM's examples are included.
  ************************************************
  */

// Prevent recursive inclusion
#ifndef __MAIN_H
#define __MAIN_H

/**
  *     IMPORTANT !     IMPORTANT !     IMPORTANT !
  *
  *     This value is incremented every time there is a change in this MCU program. Version number 1 is used until the device is taken to production. Increment should be integer 1 (i.e. the version numbers are 2, 3, 4...), no minor versions!
  */
#define VERSIONNUMBER 1
/**
  *     IMPORTANT !     IMPORTANT !     IMPORTANT !
  */


#define MAXSTRLENGTH 64         // Max length of a command
#define RMCOMMAND 0
#define RMPARAM1 1
#define RMPARAM2 2
#define RMCHECKSUM 3

// Station Commands available for PC GUI without (used without "CMD" prefix)
// Default command which tells the PC GUI that there was something wrong
#define CMDERROR 0

// Parameters: dut number (numbering from 1-, CW from top left corner), status to set
#define CMDSETDUTSTATUS 1
// Parameters: dut number; returns: status
#define CMDGETDUTSTATUS 2
// Dut status wait (default: yellow LED)
#define DSWAIT 0
// Dut status fail (default: red LED)
#define DSFAIL 1
// Dut status done (default: green LED)
#define DSDONE 2

// Parameters: state to set
#define CMDSETLOCKSTATE 3
// Returns: lock state
#define CMDGETLOCKSTATE 4
// Lock state open
#define LOCKOPEN 1
// Lock state closed
#define LOCKCLOSED 2

// Returns: lid state
#define CMDGETLIDSTATE 5
// Lid state open
#define LIDOPEN 1
// Lid state closed
#define LIDCLOSED 2

// Parameters: power state
#define CMDSETPOWER 6
// Returns: power state
#define CMDGETPOWER 7
// Power on
#define PWRON 1
// Power off
#define PWROFF 2

// Read voltage from dut sense
#define CMDDUTSENSE 8

// Gets version of the MCU software
#define CMDGETVERSION 9

// Gets into infinite loop. Used to test the watchdog functionality
#define CMDDOFAIL 10

// Set to FALSE if watchdog should be disabled
#define ENABLEWATCHDOG TRUE

// 'Main' functions
void main();
void Init(bool EnableWatchdog);
void CheckLidState();
bool ReceiveCommand();
void ExecuteCommand();

// 'Helper' functions
void Delay(__IO long Time);
void SelectADCPin(int Dut);
int IntToCharArray(char* Buffer, int Value);
int CharArrayToInt(char* Arr);
int Len(char* Str);
void PrintString(char* Buffer);
bool StringsEqual(char* Str1, char* Str2);
int CalculateChecksum(char* Str);
void ExecutePowerState();
void SendMessage(char* Str);

#endif // __MAIN_H
```

C.2 Mikrokontrolleriohjelmiston lähdekoodi

```
/**
  ************************************************
  * @file    main.c
  * @author  Eppu Ainola
  * @date    2013-06-12
  * @brief   Main Program of Software Loading Station
  ************************************************
  * @attention
  *     This program (more specifically the included init files) are from STMicroelectronics' (STM) STM32_USB-FS-Device_Lib_V4.0.0 VirtualComport_Loopback. In addition this program may include init procedures from STM's examples.
  ************************************************
  */


/**
  *     IMPORTANT !     IMPORTANT !     IMPORTANT !
  *
  *     If this code is changed after taking it to production
  *     remember to increment version number in the header!
  *
  *     IMPORTANT !     IMPORTANT !     IMPORTANT !
  */


// Includes
#include "hw_config.h"
#include "usb_lib.h"
#include "usb_desc.h"
#include "usb_pwr.h"
#include "main.h"
#include "stm32f30x_tim.h"
#include "stm32f30x_iwdg.h"
#include "stm32_it.h"


// Variables based on STM's examples.
```

```
// The naming scheme may differ from other variables because external functions may refer to these variables by these names.
__IO int ADC1ConvertedValue = 0, ADC1ConvertedVoltage = 0, CalibrationValue = 0;
__IO long TimingDelay = 0;
__IO uint32_t LsiFreq = 42000;
extern uint16_t CaptureNumber;
extern __IO uint8_t Receive_Buffer[64];
extern __IO uint32_t Receive_length;

int Send_Buffer[64];
int packet_sent=1;
int packet_receive=1;

// Various InitStructures
ADC_InitTypeDef ADCInitStructure;
ADC_CommonInitTypeDef ADCCommonInitStructure;
GPIO_InitTypeDef GPIOInitStructure;


// Buffer parameters for waiting for commands which may arrive little by little
char CmdBuffer[MAXSTRLENGTH];
char Param1Buffer[MAXSTRLENGTH];
char Param2Buffer[MAXSTRLENGTH];
char ChecksumBuffer[MAXSTRLENGTH];
int CmdBufferIndex = 0;
int Param1BufferIndex = 0;
int Param2BufferIndex = 0;
int ChecksumBufferIndex = 0;
// CmdReadMode indicates which buffer is to be appended
int CmdReadMode = 0;
bool CmdError = FALSE;

/*// If command is received, it is saved here. 0 means undefined.
int CmdID = 0;
int CmdParam1 = 0;
int CmdParam2 = 0;*/

// DUT LEDs are in arrays for easy accessability. Values 1-6 are in use.
// DUTLEDPort and DUTLEDPin are accessed as ARR[DUTNUMBER][DUTSTATUS]
GPIO_TypeDef* DUTLEDPort[7][3] = {
    {0,0,0},
    {GPIOD,GPIOD,GPIOD},
    {GPIOC,GPIOD,GPIOC},
    {GPIOD,GPIOD,GPIOD},
    {GPIOD,GPIOD,GPIOD},
    {GPIOC,GPIOD,GPIOD},
    {GPIOD,GPIOC,GPIOC}
};
const int DUTLEDPin[7][3] = {
    {0,0,0},
    {GPIO_Pin_12,GPIO_Pin_14,GPIO_Pin_13},
    {GPIO_Pin_7,GPIO_Pin_15,GPIO_Pin_6},
    {GPIO_Pin_10,GPIO_Pin_11,GPIO_Pin_6},
    {GPIO_Pin_3,GPIO_Pin_5,GPIO_Pin_4},
    {GPIO_Pin_11,GPIO_Pin_0,GPIO_Pin_2},
    {GPIO_Pin_1,GPIO_Pin_12,GPIO_Pin_10}
};

// Lock pin
GPIO_TypeDef* LockPort = GPIOB;
int LockPin = GPIO_Pin_1;

// DUT Power pin
GPIO_TypeDef* DUTPowerPort = GPIOE;
int DUTPowerPin = GPIO_Pin_7;

// Various states in station
bool LockOpen = FALSE;
bool LockOpenWhenPowered = FALSE; // NOTICE! if solenoid is operated in reverse manner (powering solenoid releases lock), set this variable TRUE!
bool LidOpen = FALSE;
bool NeedlePowerOn = FALSE;
int DutStatus[] = { 0, DSWAIT, DSWAIT, DSWAIT, DSWAIT, DSWAIT, DSWAIT }; // Dut indexes 1-6


// ADC pins are in arrays for easy accessability. dut indexes are 1-6.
GPIO_TypeDef* ADCPORT[] = {0, GPIOF, GPIOC, GPIOC, GPIOC, GPIOC, GPIOA};
int ADCPIN[] = {0, GPIO_Pin_2, GPIO_Pin_2, GPIO_Pin_0, GPIO_Pin_1, GPIO_Pin_3, GPIO_Pin_1};
int ADCCHANNEL[] = {0, ADC_Channel_10, ADC_Channel_8, ADC_Channel_6, ADC_Channel_7, ADC_Channel_9, ADC_Channel_2};


/**
  * @brief      Main program. Contains inits and while loop which includes command and lid open checking.
  * @param      None.
  * @retval     None.
  */
void main()
{
  // Init everything
  Init(ENABLEWATCHDOG);

  // Main program which runs indefinately
  while (TRUE)
  {
    IWDG_ReloadCounter();

    // Device will be configured only when it is connected to PC
    if (bDeviceState == CONFIGURED)
    {
      // Check if a command has been received
      if (ReceiveCommand())
      {
        // Command has been received successfully, execute it
        ExecuteCommand();
      }

      // Check if lid is open (and if so, disable DUT power) regardless of USB connection (usually if USB cable is disconnected, MCU is unpowered, but there is a chance that the connection is 'broken' even if the cable would be attached)
      CheckLidState();
    }
}


/**
  * @brief      Init program. Initiates all required internal peripherals. May contain direct code snippets from STM's examples.
  * @param      EnableWatchdog: True to enable watchdog, false to disable. May be convenient to disable when debugging.
  * @retval     None.
  */
void Init(bool EnableWatchdog)
{
  // Setup SysTick Timer for 1 msec interrupts
  // If usec resolution is wished, change divider 1000 to 1000000. BUT remember to adjust watchdog accordingly.
  SysTick_Config(SystemCoreClock / 1000);


  // Init LEDs for debugging
  STM_EVAL_LEDInit(LED3); // DUT 1 WAIT
  STM_EVAL_LEDInit(LED4); // DUT 1 FAIL
  STM_EVAL_LEDInit(LED5); // DUT 1 DONE
  STM_EVAL_LEDInit(LED6); // LID OPEN
  STM_EVAL_LEDInit(LED7); // LOCK POWER ON
  STM_EVAL_LEDInit(LED8); // NEEDLE POWER ON
  STM_EVAL_LEDInit(LED9); // WATCHDOG (has resetted device)


  // Setup watchdog
  if (RCC_GetFlagStatus(RCC_FLAG_IWDGRST) != RESET)
  {
    // If watchdog resetted the system, light up a led
    STM_EVAL_LEDOn(LED9);
    RCC_ClearFlag();
  }
  else
  {
    // Watchdog didnt initiate the last reset
    STM_EVAL_LEDOff(LED9);
  }

  // Configure low speed timer for watchdog
  TIM16_ConfigForLSI();
  while(CaptureNumber != 2);
  TIM_ITConfig(TIM16, TIM_IT_CC1, DISABLE);

  /* Set counter reload value to obtain 250ms IWDG TimeOut.
    Counter Reload Value = 250ms * IWDG counter clock period
                         = 250ms * (LSI/32)
```

```
                          = 0.25s * (LsiFreq/32)
                          = LsiFreq / (32 * 4)
                          = LsiFreq / 128
 */
IWDG_WriteAccessCmd(IWDG_WriteAccess_Enable);
IWDG_SetPrescaler(IWDG_Prescaler_32);
IWDG_SetReload(LsiFreq/128);
IWDG_ReloadCounter();

// Enable IWDG if the function parameter is true
if (EnableWatchdog)
{
  IWDG_Enable();
}


// Init USB as Virtual COM Port
Set_System();
Set_USBClock();
USB_Interrupts_Config();
USB_Init();

// Init various clocks/timers
RCC_ADCCLKConfig(RCC_ADC12PLLCLK_Div2);
RCC_AHBPeriphClockCmd(RCC_AHBPeriph_ADC12 | RCC_AHBPeriph_GPIOB | RCC_AHBPeriph_GPIOC | RCC_AHBPeriph_GPIOD | RCC_AHBPeriph_GPIOE, ENABLE);


/**
  * Init ADC pins
  */

ADC_StructInit(&ADCInitStructure);
ADC_VoltageRegulatorCmd(ADC1, ENABLE);
// Insert delay equal to 1 ms (documentation: 10 us is enough)
Delay(1);
ADC_SelectCalibrationMode(ADC1, ADC_CalibrationMode_Single);
ADC_StartCalibration(ADC1);
while(ADC_GetCalibrationStatus(ADC1) != RESET);
CalibrationValue = ADC_GetCalibrationValue(ADC1);

ADCCommonInitStructure.ADC_Mode = ADC_Mode_Independent;
ADCCommonInitStructure.ADC_Clock = ADC_Clock_AsynClkMode;
ADCCommonInitStructure.ADC_DMAAccessMode = ADC_DMAAccessMode_Disabled;
ADCCommonInitStructure.ADC_DMAMode = ADC_DMAMode_OneShot;
ADCCommonInitStructure.ADC_TwoSamplingDelay = 0;
ADC_CommonInit(ADC1, &ADCCommonInitStructure);

ADCInitStructure.ADC_ContinuousConvMode = ADC_ContinuousConvMode_Enable;
ADCInitStructure.ADC_Resolution = ADC_Resolution_12b;
ADCInitStructure.ADC_ExternalTrigConvEvent = ADC_ExternalTrigConvEvent_0;
ADCInitStructure.ADC_ExternalTrigEventEdge = ADC_ExternalTrigEventEdge_None;
ADCInitStructure.ADC_DataAlign = ADC_DataAlign_Right;
ADCInitStructure.ADC_OverrunMode = ADC_OverrunMode_Disable;
ADCInitStructure.ADC_AutoInjMode = ADC_AutoInjec_Disable;
ADCInitStructure.ADC_NbrOfRegChannel = 1;
ADC_Init(ADC1, &ADCInitStructure);

// Default ADC Pin is DUT1's ADC pin
SelectADCPin(1);

// Start ADC conversion
ADC_Cmd(ADC1, ENABLE);
while(!ADC_GetFlagStatus(ADC1, ADC_FLAG_RDY));
ADC_StartConversion(ADC1);


/**
  * Init all GPIOs
  */

GPIOInitStructure.GPIO_Pin = GPIO_Pin_0 | GPIO_Pin_1 | GPIO_Pin_2;
GPIOInitStructure.GPIO_Mode = GPIO_Mode_OUT;
GPIOInitStructure.GPIO_OType = GPIO_OType_PP;
GPIOInitStructure.GPIO_Speed = GPIO_Speed_50MHz;
GPIOInitStructure.GPIO_PuPd = GPIO_PuPd_NOPULL;
GPIO_Init(GPIOB, &GPIOInitStructure);
GPIO_ResetBits(GPIOB, GPIOInitStructure.GPIO_Pin);


GPIOInitStructure.GPIO_Pin = GPIO_Pin_6 | GPIO_Pin_7 | GPIO_Pin_10 | GPIO_Pin_11 | GPIO_Pin_12;
GPIOInitStructure.GPIO_Mode = GPIO_Mode_OUT;
GPIOInitStructure.GPIO_OType = GPIO_OType_PP;
GPIOInitStructure.GPIO_Speed = GPIO_Speed_50MHz;
GPIOInitStructure.GPIO_PuPd = GPIO_PuPd_NOPULL;
GPIO_Init(GPIOC, &GPIOInitStructure);
GPIO_ResetBits(GPIOC, GPIOInitStructure.GPIO_Pin);

GPIOInitStructure.GPIO_Pin = GPIO_Pin_0 | GPIO_Pin_1 | GPIO_Pin_2 | GPIO_Pin_3 | GPIO_Pin_4 | GPIO_Pin_5 | GPIO_Pin_6 | GPIO_Pin_10 | GPIO_Pin_11 | GPIO_Pin_12 | GPIO_Pin_13 | GPIO_Pin_14 | GPIO_Pin_15;
GPIOInitStructure.GPIO_Mode = GPIO_Mode_OUT;
GPIOInitStructure.GPIO_OType = GPIO_OType_PP;
GPIOInitStructure.GPIO_Speed = GPIO_Speed_50MHz;
GPIOInitStructure.GPIO_PuPd = GPIO_PuPd_NOPULL;
GPIO_Init(GPIOD, &GPIOInitStructure);
GPIO_ResetBits(GPIOD, GPIOInitStructure.GPIO_Pin);

GPIOInitStructure.GPIO_Pin = GPIO_Pin_7;
GPIOInitStructure.GPIO_Mode = GPIO_Mode_OUT;
GPIOInitStructure.GPIO_OType = GPIO_OType_PP;
GPIOInitStructure.GPIO_Speed = GPIO_Speed_50MHz;
GPIOInitStructure.GPIO_PuPd = GPIO_PuPd_NOPULL;
GPIO_Init(GPIOE, &GPIOInitStructure);
GPIO_ResetBits(GPIOE, GPIOInitStructure.GPIO_Pin);

// Inputs
GPIOInitStructure.GPIO_Pin = GPIO_Pin_4 | GPIO_Pin_5;
GPIOInitStructure.GPIO_Mode = GPIO_Mode_IN;
GPIOInitStructure.GPIO_PuPd = GPIO_PuPd_NOPULL;
GPIO_Init(GPIOC, &GPIOInitStructure);


// Init lid state
int lidsw1state = GPIO_ReadInputDataBit(GPIOC, GPIO_Pin_5);
int lidsw2state = GPIO_ReadInputDataBit(GPIOC, GPIO_Pin_4);
LidOpen = (lidsw1state + lidsw2state > 0) ? TRUE : FALSE;

}


/**
  * @brief      Check that when lid is open, disable power for user safety.
  * @param      None
  * @retval     None
  */
void CheckLidState()
{
  // States are 0 when GND and 1 when Vdd
  int lidsw1state = GPIO_ReadInputDataBit(GPIOC, GPIO_Pin_5);
  int lidsw2state = GPIO_ReadInputDataBit(GPIOC, GPIO_Pin_4);
  // Check if this means that the lid's state has changed
  // (open -> closed) || (closed -> open)
  if (LidOpen && lidsw1state + lidsw2state == 0 || !LidOpen && lidsw1state + lidsw2state > 0)
  {
    // Simple debounce
    Delay(100); // time in ms

    // Check the lid state again for certainly stable signal
    // States are 0 when GND and 1 when Vdd
    lidsw1state = GPIO_ReadInputDataBit(GPIOC, GPIO_Pin_5);
    lidsw2state = GPIO_ReadInputDataBit(GPIOC, GPIO_Pin_4);

    LidOpen = (lidsw1state + lidsw2state > 0) ? TRUE : FALSE;
  }

  // If lid is really open, shut down the power. if the state did not change, the power state does not change
  // This check needs to take place on every loop in case of accidental powering on elsewhere in code
  if (LidOpen)
  {
    NeedlePowerOn = FALSE;
    ExecutePowerState();
  }
```

```

  // Toggle LED for debugging. Led ON means that the lid is open
  STM_EVAL_LEDOff(LED6);
  if (LidOpen) STM_EVAL_LEDOn(LED6);
}


/**
  * @brief      Read and decipher a received command. The command may be received in pieces
  *             and thus the buffers may have to be appended later.
  * @param      None
  * @retval     TRUE if command is ready to be executed, FALSE if command is not ready yet
  */
bool ReceiveCommand()
{
  CDC_Receive_DATA();
  if (Receive_length  != 0)
  {
    for (int i = 0; i < Receive_length; i++)
    {
      // Special char '<' OR command max length is exceeded (starts a new command)
      if (Receive_Buffer[i] == '<' || CmdBufferIndex >= MAXSTRLENGTH)
      {
        CmdBuffer[0] = '\0';
        Param1Buffer[0] = '\0';
        Param2Buffer[0] = '\0';
        ChecksumBuffer[0] = '\0';
        CmdBufferIndex = 0;
        Param1BufferIndex = 0;
        Param2BufferIndex = 0;
        ChecksumBufferIndex = 0;
        CmdReadMode = 0;
        CmdError = FALSE;
      }

      // Special char '>' (ends a command - indicates it is ready to be executed)
      else if (Receive_Buffer[i] == '>')
      {
        ChecksumBuffer[ChecksumBufferIndex] = '\0';

        // Check that the received message has not been corrupted (8-bit checksum)
        // For successful transmission, the following holds:
        //      SUM{Param1 (bytewise), Param2 (bytewise), Checksum (as single byte)} & 0xFF == 0
        int checksum = CharArrayToInt(ChecksumBuffer);
        for (int j = 0; j < Param1BufferIndex; j++) checksum += Param1Buffer[j];
        for (int j = 0; j < Param2BufferIndex; j++) checksum += Param2Buffer[j];

        checksum &= 0xFF;

        // Incorrect checksum
        if (checksum != 0)
        {
          CmdError = TRUE;
        }

        // Reset command buffer when a command is to be executed. this way entering > does not trigger unwanted actions
        Receive_length = 0;
        CmdBufferIndex = 0;
        return TRUE;
      }

      // Special char ':' changes read mode
      else if (Receive_Buffer[i] == ':')
      {
        switch (CmdReadMode)
        {
        case RMCOMMAND:
          CmdBuffer[CmdBufferIndex] = '\0';
          CmdReadMode = RMPARAM1;
          break;
        case RMPARAM1:
          Param1Buffer[Param1BufferIndex] = '\0';
          CmdReadMode = RMPARAM2;
          break;
        default:
          // Do nothing
          break;
        }
      }

      // Special char '|' changed read mode to checksum
      else if (Receive_Buffer[i] == '|')
      {
        switch (CmdReadMode)
        {
        case RMPARAM1:
          Param1Buffer[Param1BufferIndex] = '\0';
          break;
        case RMPARAM2:
          Param2Buffer[Param2BufferIndex] = '\0';
          break;
        default:
          // Do nothing
          break;
        }
        CmdReadMode = RMCHECKSUM;
      }

      // Else append char to correct variable
      else
      {
        switch (CmdReadMode)
        {
        case RMCOMMAND:
          CmdBuffer[CmdBufferIndex] = Receive_Buffer[i];
          CmdBufferIndex++;
          break;
        case RMPARAM1:
          Param1Buffer[Param1BufferIndex] = Receive_Buffer[i];
          Param1BufferIndex++;
          break;
        case RMPARAM2:
          Param2Buffer[Param2BufferIndex] = Receive_Buffer[i];
          Param2BufferIndex++;
          break;
        case RMCHECKSUM:
          ChecksumBuffer[ChecksumBufferIndex] = Receive_Buffer[i];
          ChecksumBufferIndex++;
          break;
        default:
          // Do nothing
          break;
        }
      }
    }

    // Reset Receive_length when all has been received
    Receive_length = 0;
  }

  // Default return, FALSE, means that the command has not been received completely yet
  return FALSE;
}


/**
  * @brief      Executes the command that was received with ReceiveCommand().
  * @param      None
  * @retval     None
  */
void ExecuteCommand()
{
  int Status, CmdParam1, CmdParam2;

  /**
    * Execute the command that was received.
    */
  if (CmdError)
  {
    SendMessage("ERROR_CHECKSUM");
  }
  else if (StringsEqual(CmdBuffer, "SETDUTSTATUS"))
  {
    // Parse Param2
    CmdParam2 = CharArrayToInt(Param2Buffer);

    // Parse Param1
    if (StringsEqual(Param1Buffer, "WAIT")) CmdParam1 = DSWAIT;
    else if (StringsEqual(Param1Buffer, "FAIL")) CmdParam1 = DSFAIL;
    else if (StringsEqual(Param1Buffer, "DONE")) CmdParam1 = DSDONE;

    DutStatus[CmdParam2] = CmdParam1;
```

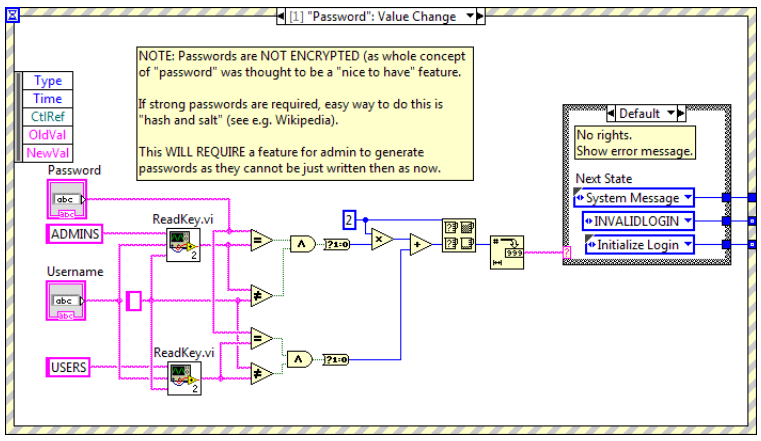
```

    // Reset all LEDs for the given DUT
    for (int i = 0; i < 3; i++) GPIO_ResetBits(DUTLEDPort[CmdParam2][i], DUTLEDPin[CmdParam2][i]);

    // Light up the given LED on given DUT
    GPIO_SetBits(DUTLEDPort[CmdParam2][CmdParam1], DUTLEDPin[CmdParam2][CmdParam1]);

    // Toggle led for debugging (DUT 1)
    if (CmdParam2 == 1)
    {
      STM_EVAL_LEDOff(LED3);
      STM_EVAL_LEDOff(LED4);
      STM_EVAL_LEDOff(LED5);
      switch(CmdParam1)
      {
        case DSWAIT: STM_EVAL_LEDOn(LED3); break;
        case DSFAIL: STM_EVAL_LEDOn(LED4); break;
        case DSDONE: STM_EVAL_LEDOn(LED5); break;
        default: break; // Do nothing
      }
    }
  }
  else if (StringsEqual(CmdBuffer, "GETDUTSTATUS"))
  {
    CmdParam1 = CharArrayToInt(Param1Buffer);
    Status = DutStatus[CmdParam1];
    switch(Status)
    {
    case DSWAIT: SendMessage("WAIT"); break;
    case DSFAIL: SendMessage("FAIL"); break;
    case DSDONE: SendMessage("DONE"); break;
    default:
      // Do nothing
      break;
    }
  }
  else if (StringsEqual(CmdBuffer, "SETLOCKSTATE"))
  {
    // Set status variable
    if (StringsEqual(Param1Buffer, "OPEN"))
    {
      LockOpen = TRUE;
    }
    else if (StringsEqual(Param1Buffer, "CLOSED"))
    {
      LockOpen = FALSE;
    }

    // Execute correct state
    // Lock power on (default: powered on when closed UNLESS LockOpenWhenPowered == TRUE)
    if (LockOpen == LockOpenWhenPowered)
    {
      GPIO_SetBits(LockPort, LockPin);
    }
    // Lock power off
    else
    {
      GPIO_ResetBits(LockPort, LockPin);
    }

    // Toggle LED for debugging
    STM_EVAL_LEDOff(LED7);
    if (LockOpen) STM_EVAL_LEDOn(LED7);
  }
  else if (StringsEqual(CmdBuffer, "GETLOCKSTATE"))
  {
    if (LockOpen)
    {
      SendMessage("OPEN");
    }
    else
    {
      SendMessage("CLOSED");
    }
  }
  else if (StringsEqual(CmdBuffer, "GETLIDSTATE"))
  {
    if (LidOpen)
    {
      SendMessage("OPEN");
    }
    else
    {
      SendMessage("CLOSED");
    }
  }
  else if (StringsEqual(CmdBuffer, "SETPOWER"))
  {
    // Set status variable. If lid is open, power cannot be set on
    if (StringsEqual(Param1Buffer, "OFF") || LidOpen)
    {
      NeedlePowerOn = FALSE;
    }
    else if (StringsEqual(Param1Buffer, "ON"))
    {
      NeedlePowerOn = TRUE;
    }

    // Execute correct state
    ExecutePowerState();

    // Toggle LED for debugging
    STM_EVAL_LEDOff(LED8);
    if (NeedlePowerOn) STM_EVAL_LEDOn(LED8);
  }
  else if (StringsEqual(CmdBuffer, "GETPOWER"))
  {
    if (NeedlePowerOn)
    {
      SendMessage("ON");
    }
    else
    {
      SendMessage("OFF");
    }
  }
  else if (StringsEqual(CmdBuffer, "DUTSENSE"))
  {
    SelectADCPin(CharArrayToInt(Param1Buffer));

    // ADC value seems to give a correct value only on second read. Do a dummy read first.

    while(ADC_GetFlagStatus(ADC1, ADC_FLAG_EOC) == RESET);
    ADC1ConvertedValue = ADC_GetConversionValue(ADC1);
    Delay(1);

    // Then the actual read

    // Read ADC
    while(ADC_GetFlagStatus(ADC1, ADC_FLAG_EOC) == RESET);
    // Get ADC1 converted data
    ADC1ConvertedValue = ADC_GetConversionValue(ADC1);
    // Compute the voltage. 3000 is used to scale the measured value to voltages
    ADC1ConvertedVoltage = (ADC1ConvertedValue * 3000) / 0xFFF;

    char Voltagestr[10];
    IntToCharArray(Voltagestr, ADC1ConvertedVoltage);
    SendMessage(Voltagestr);

    // Without delay ADC seems to hang especially if slower ADC conversions are used
    Delay(1);
  }
  else if (StringsEqual(CmdBuffer, "GETVERSION"))
  {
    // Print the version number. Version is stored in the header file
    int Version = VERSIONNUMBER;
    char VersionStr[MAXSTRLENGTH];
    IntToCharArray(VersionStr, Version);
    SendMessage(VersionStr);
  }
  else if (StringsEqual(CmdBuffer, "DOFAIL"))
  {
    // Go to infinite loop on purpose. Watchdog should reset the device
    while (TRUE);
  }
  else // No matches
  {
    SendMessage("ERROR_NOCOMMANDMATCH");
  }
}



/**
 *
 *      HELPER FUNCTIONS BELOW THIS
 *
 */

```

```

/**
  * @brief      Inserts a delay time. This function is from STM's examples
  * @param      Time: specifies the delay time length, in milliseconds.
  * @retval     None
  */
void Delay(__IO long Time)
{
  TimingDelay = Time;
  while(TimingDelay != 0);
}


/**
  * @brief      Selects (changes) pin from which to read the ADC signal.
  * @param      Dut: specifies the device under test to read (integers 1-6).
  * @retval     None
  */
void SelectADCPin(int Dut)
{
  GPIOInitStructure.GPIO_Pin = ADCPIN[Dut];
  GPIOInitStructure.GPIO_Mode = GPIO_Mode_AN;
  GPIOInitStructure.GPIO_PuPd = GPIO_PuPd_NOPULL;
  GPIO_Init(ADCPORT[Dut], &GPIOInitStructure);

  // NOTICE! If ADC hangs, add Delay or change ADC_SampleTime_7Cycles5 -> ADC_SampleTime_1Cycles5
  ADC_RegularChannelConfig(ADC1, ADCCHANNEL[Dut], 1, ADC_SampleTime_1Cycles5);
}


/**
  * @brief      Convert a integer to ASCII char array.
  * @param      Arr: specifies the array into which the result is saved into. Caller is responsible to reserve enough space even for \0.
  * @param      Value: specifies the integer which is to be converted (value limited to 32 bits).
  * @retval     Returns length of the resulting string OR -1 in case of error.
  */
int IntToCharArray(char* Arr, int Value)
{
  // NOTICE! This program does not require negative values to be converted, so it has not been implemented! Implement if required.
  if (Value < 0)
  {
    Arr = "-1";
    return -1;
  }


  int Index = 0;
  int Divider = 1000000000;

  while (Divider > 0)
  {
    // Convert to char if
    // a) value is larger than divider (i.e. this is the most significant digit)
    // b) the largest significant digit is already found (otherwise algorithm would skip 0s within the value!)
    // c) if the value is just 0, convert it
    if (Value >= Divider || Index > 0 || (Divider == 1 && Value == 0))
    {
      // NOTICE: if the next two lines are reversed (of course with correct array indexing), the compiler seems to ignore the "index++" line.
      Index++;
      Arr[Index-1] = '0' + Value / Divider;

      Value %= Divider;
    }

    Divider /= 10;
  }

  // Add \0 to indicate end of string
  Arr[Index] = '\0';

  return Index;
}


/**
  * @brief      Convert an ASCII char array to integer. Does NOT check validity of the char array.
  * @param      Arr: specifies the array from where the string is read from. End of array should be indicated with \0.
  * @retval     Returns result of the conversion (at most 32-bit value).
  */
int CharArrayToInt(char* Arr)
{
  int Result = 0;
  int Multiplier = 1;
  int ArrLen = Len(Arr);

  for (int i = ArrLen - 1; i >= 0; i--)
  {
    // -'0' is a way to convert a char representing a number to an actual number
    Result += (Arr[i] - '0') * Multiplier;

    Multiplier *= 10;
  }

  return Result;
}


/**
  * @brief      Calculates length of a string.
  * @param      Str: string of interest.
  * @retval     Returns length of the str parameter.
  */
int Len(char* Str)
{
  int i;
  for (i = 0; Str[i] != '\0'; i++);
  return i;
}


/**
  * @brief      Prints a char array to the COM port (USB) appended by \r\n for readability (software should ignore these but a human may find them useful while debugging).
  * @param      Buffer: array to be sent. Max size is VIRTUAL_COM_PORT_DATA_SIZE - 2 (= 62) characters. Caller is responsible for providing two empty bytes in the end of the buffer to be printed.
  * @retval     None
  */
void PrintString(char* Buffer)
{
  int BufferLen = Len(Buffer);
  if (BufferLen < VIRTUAL_COM_PORT_DATA_SIZE)
  {
    // Wait until the previous packet was sent
    while (packet_sent != 1);

    // Append \r\n
    Buffer[BufferLen] = '\r';
    BufferLen++;
    Buffer[BufferLen] = '\n';
    BufferLen++;

    // Send the new string
    CDC_Send_DATA((unsigned char*)Buffer, BufferLen);
  }
}


/**
  * @brief      Compares two equal strings (char*).
  * @param      Str1: First string.
  * @param      Str2: Second string.
  * @retval     Returns TRUE if the strings equal, FALSE if not.
  */
bool StringsEqual(char* Str1, char* Str2)
{
  int i = 0;
  while (Str1[i] != '\0' && Str2[i] != '\0')
  {
    if (Str1[i] != Str2[i])
    {
      return FALSE;
    }
    i++;
```

```
823   }
824
825   // If other string is about to continue, they are not equal.
826   if (Str1[i] != '\0' || Str2[i] != '\0')
827   {
828     return FALSE;
829   }
830
831   return TRUE;
832 }
833
834
835 /**
836   * @brief      Calculates one byte checksum for a message. (2-complement of byte-wise sum)
837   * @param      Str: String for which to calculate checksum.
838   * @retval     Returns unsigned 8-bit value which represents the checksum.
839   */
840 int CalculateChecksum(char* Str)
841 {
842   int Checksum = 0;
843   for (int i = 0; Str[i] != '\0'; i++)
844   {
845     Checksum += Str[i];
846   }
847
848   // 2's complement
849   Checksum = ~Checksum;
850   Checksum++;
851
852   // Take only the LSB
853   Checksum &= 0xFF;
854
855   return Checksum;
856 }
857
858
859 /**
860   * @brief      Executes power on or power down according to the state variable.
861   * @param      None
862   * @retval     None
863   */
864 void ExecutePowerState()
865 {
866   // DUT power on
867   if (NeedlePowerOn)
868   {
869     GPIO_SetBits(DUTPowerPort, DUTPowerPin);
870   }
871   // DUT power off
872   else
873   {
874     GPIO_ResetBits(DUTPowerPort, DUTPowerPin);
875   }
876 }
877
878
879 /**
880   * @brief      Sends a whole message correctly formatted.
881   * @param      Str: Payload of the message. Caller is responsible for checking that message length is at most MAXSTRLENGTH - 7! (6 because of 3 bytes for <|>\0 and at most 3 for checksum)
882   * @retval     None.
883   */
884 void SendMessage(char* Str)
885 {
886   char SendBuffer[MAXSTRLENGTH];
887
888   // Calculate checksum. MCU replies are single 'words' (no parameters) and thus checksum is applied to whole message content.
889   // Strictly speaking checksum is not necessary with these replies (because they differ quite many bits from each others), but it is not a bad thing to have extra protection.
890   int Checksum = CalculateChecksum(Str);
891
892   // Format message
893   SendBuffer[0] = '<';
894   int StrLen = Len(Str);
895   for (int i = 0; i < StrLen; i++)
896   {
897     SendBuffer[i+1] = Str[i];
898   }
899   SendBuffer[StrLen + 1] = '|';
900   int ChecksumLength = IntToCharArray(SendBuffer + StrLen + 2, Checksum);
901   SendBuffer[StrLen + 2 + ChecksumLength] = '>';
902   SendBuffer[StrLen + 2 + ChecksumLength + 1] = '\0';
903
904   PrintString(SendBuffer);
905 }
```

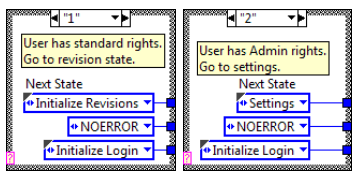
D Liite: Latausaseman käyttöliittymään liittyvä lähdekoodi

Kuvissa D.1–D.45 on listattu käyttöliittymän lähdekoodi. LabVIEW-koodi on kommentoitu lukemisen selkeyttämiseksi, koska graafisen LabVIEW-ohjelman funktiosymbolit eivät välttämättä ole itsestään selviä, jos LabVIEW ei ole ennestään tuttu. Kuvasarjan seassa on .vi-päätteisiä otsikoita. Nämä ovat niitä edeltävässä LabVIEW-koodissa käytettyjä latausaseman moduuleita. VI-tulee sanoista *virtuaalinen instrumentti (engl. virtual instrument)*. Lopuksi on C#.NET-kielinen ProgrammerLauncherin lähdekoodi. Siitä on jätetty lokitiedoston muotoilusta vastaavan `Logger`-luokan koodi pois, koska lokitiedoston yksityiskohtainen muotoilu sisältää luottamuksellista tietoa.

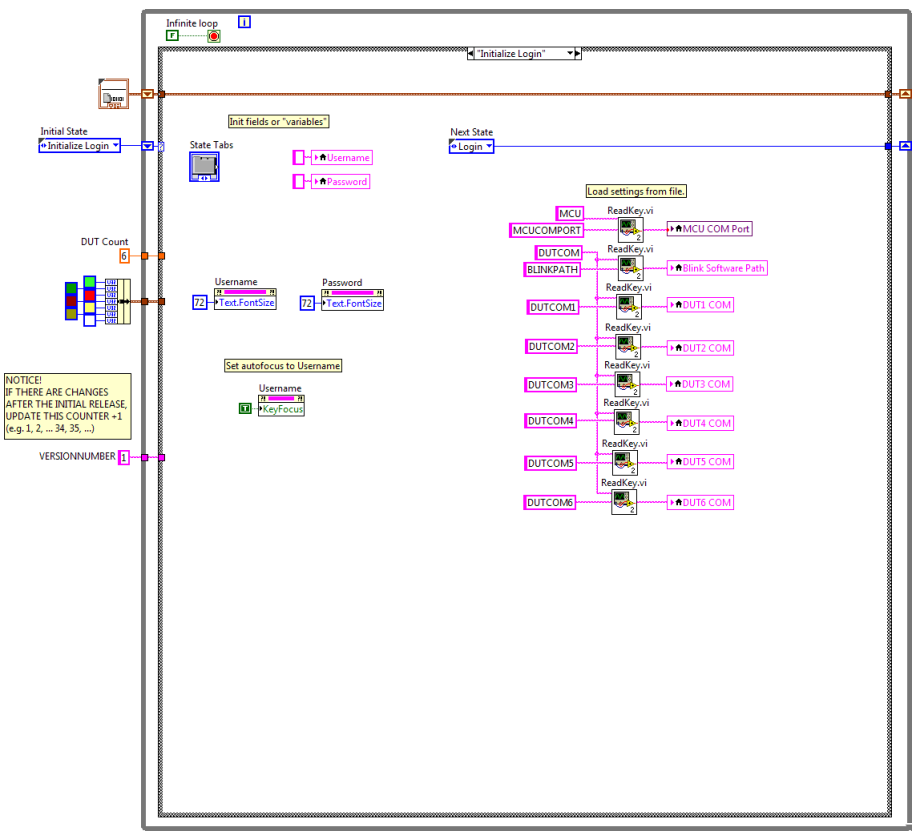


Kuva D.3: Kirjautumisikkunan tapahtumankäsittely, kun salasana syötetään.

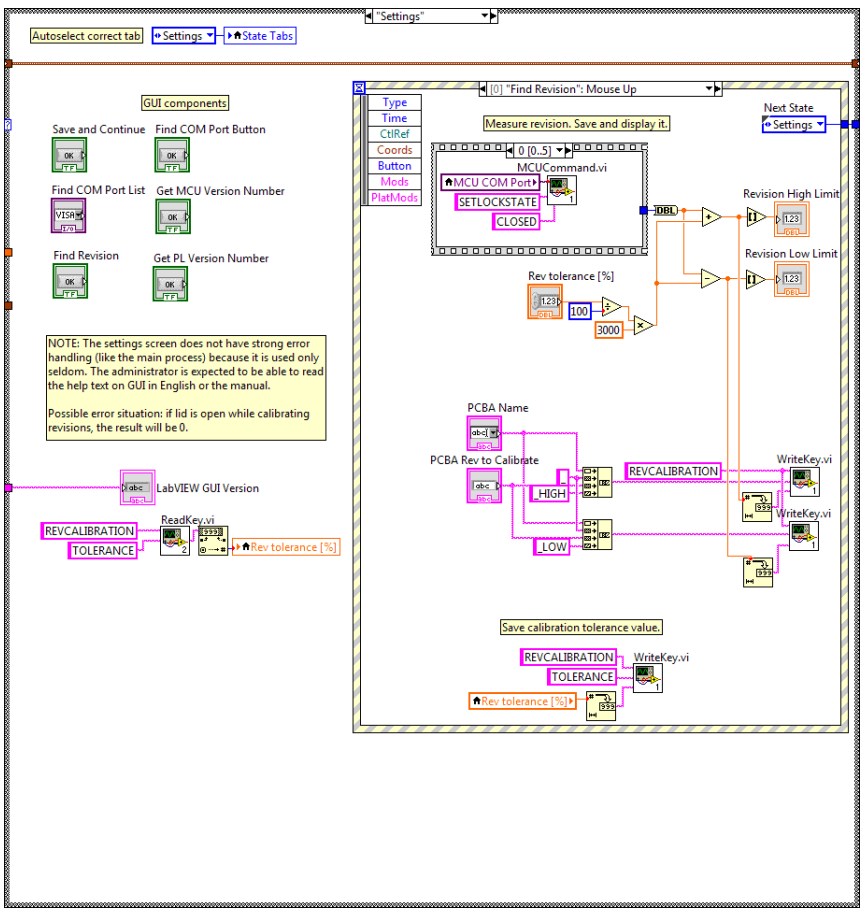
D.1 Latausaseman graafinen LabVIEW-lähdekoodi



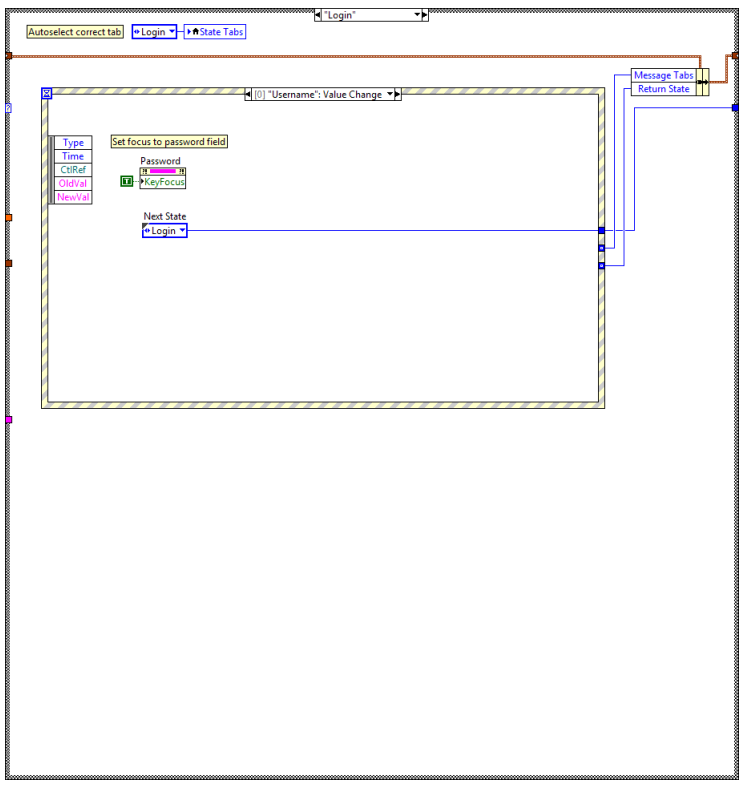
Kuva D.4: Kun syötetyllä salasanalle ja käyttäjätunnukselle löytyy operaattorioikeus, käyttäjä ohjataan revisiotilaan. Jos tunnuksille on määritetty huoltohenkilöoikeus, käyttäjä ohjataan asetustilaan.



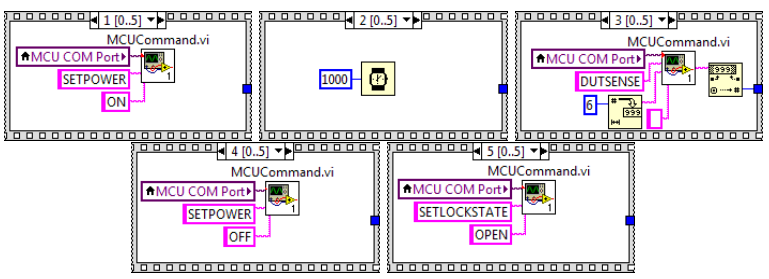
Kuva D.1: Tila 1. Kirjautumisikkunan alustus ja asetusten lukeminen tiedostosta.



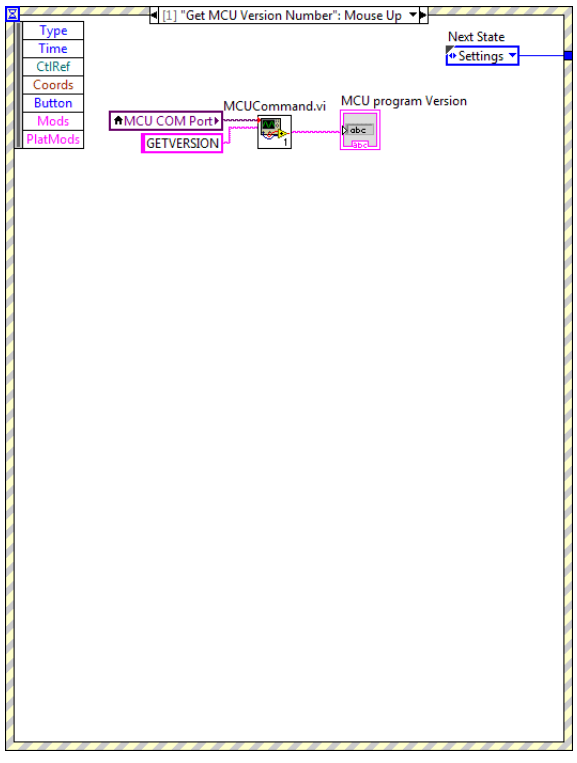
Kuva D.5: Tila 3. Asetustila.



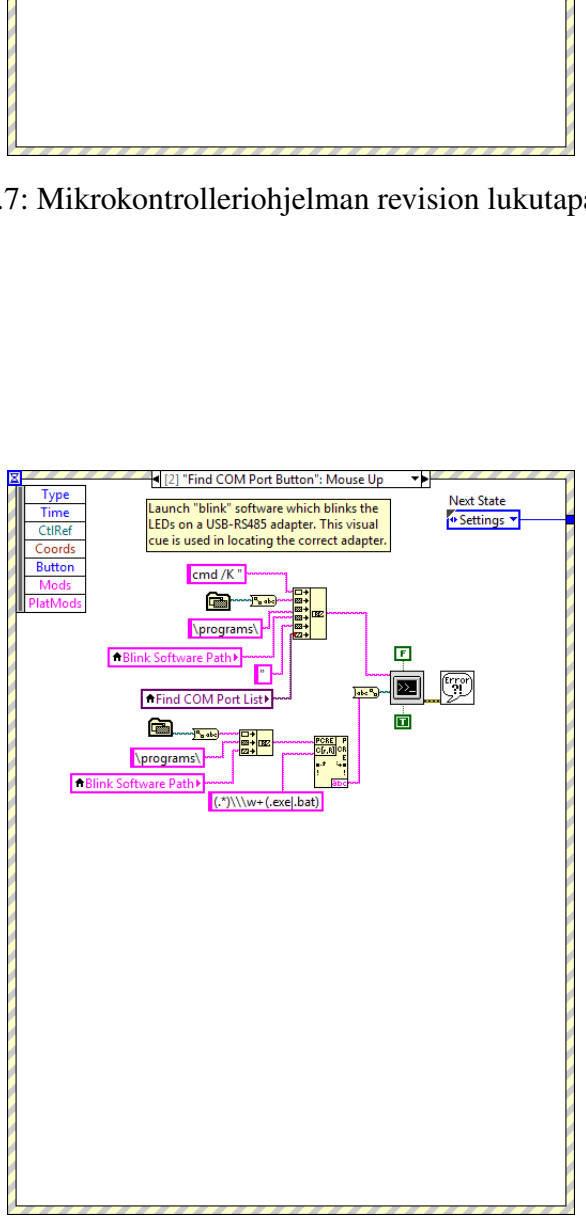
Kuva D.2: Tila 2. Kirjautumisikkuna.



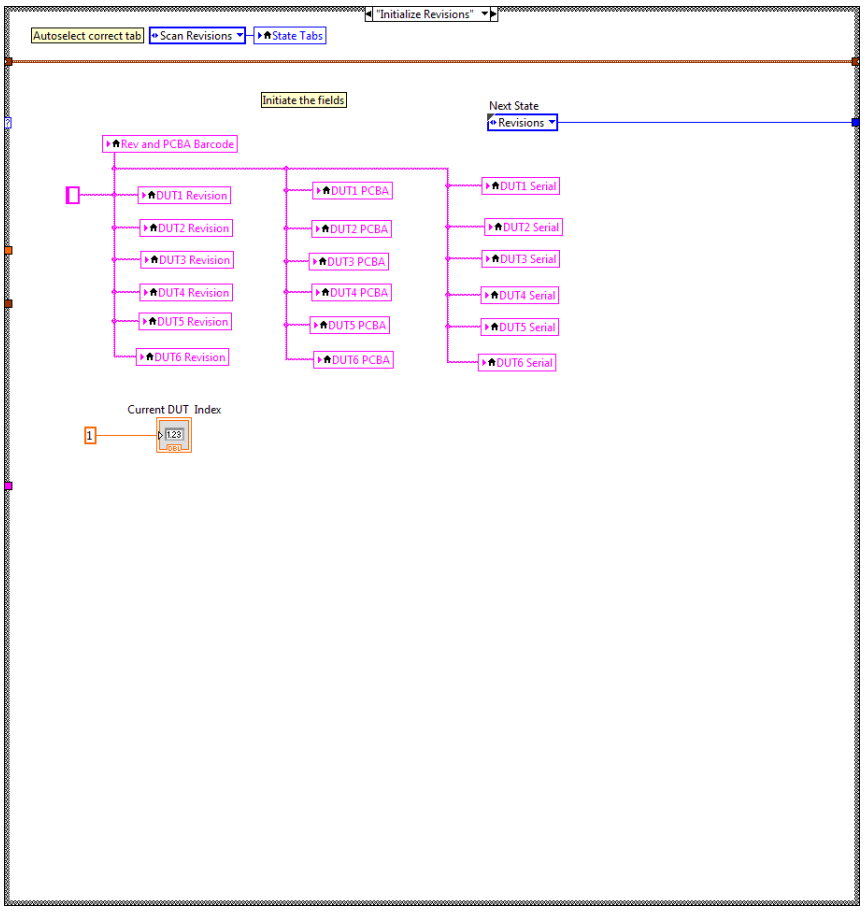
Kuva D.6: Asetustilassa revisioiden lukuun vaadittava sekvenssi, jossa asemaan asetetaan käyttöjännite, luetaan revisio ja poistetaan käyttöjännite.



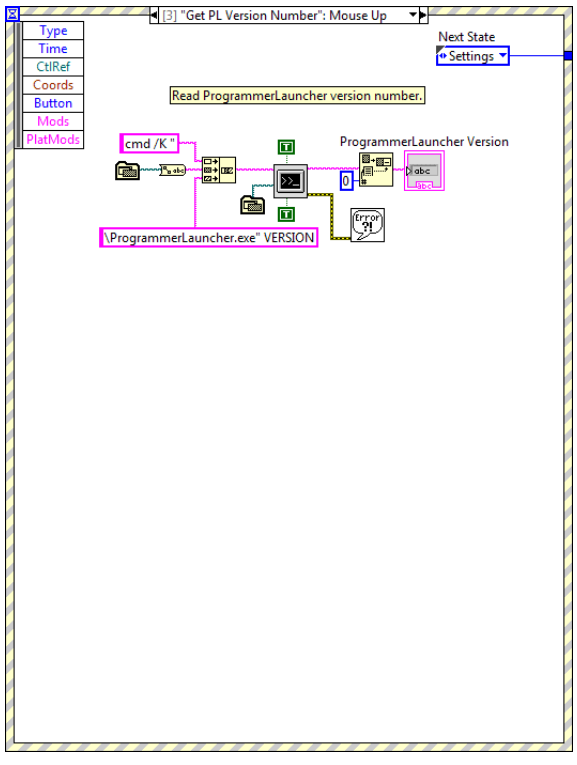
Kuva D.7: Mikrokontrolleriohjelman revision lukutapahtuma.



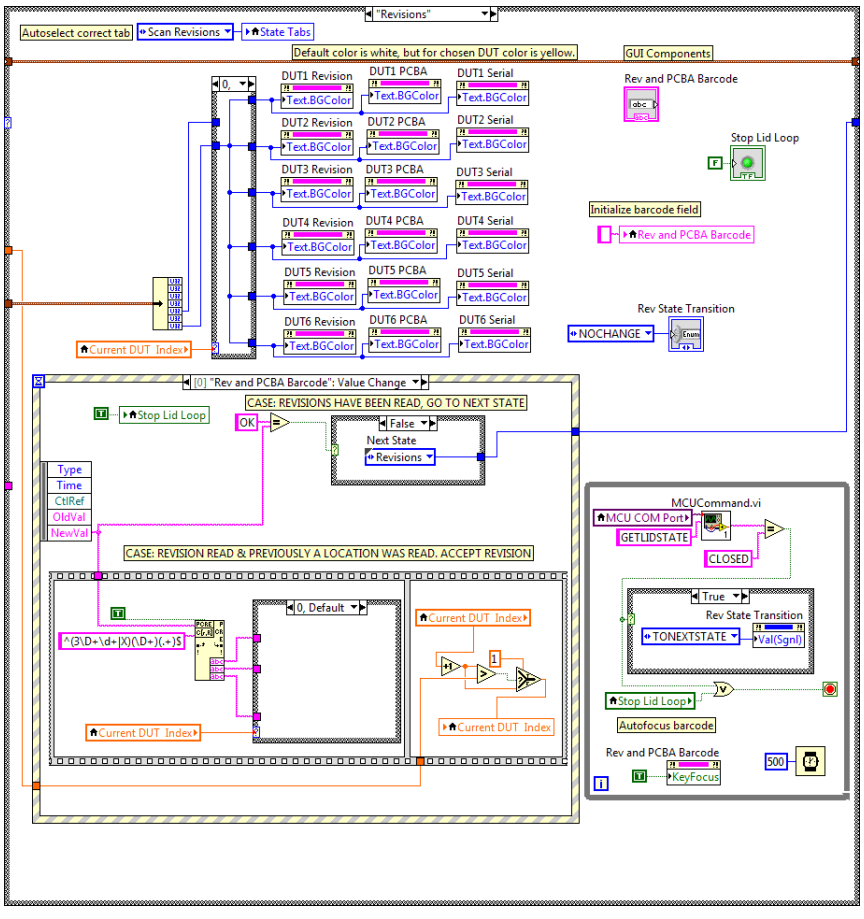
Kuva D.8: USB–TIA/EIA-485-muuntimen sarjaportin etsimistapahtuma.



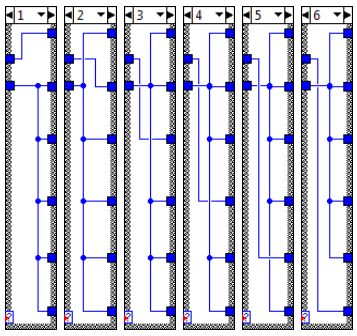
Kuva D.11: Tila 4. Revisionäkymän alustustila.



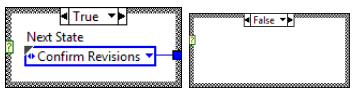
Kuva D.9: ProgrammerLauncher-ohjelman revision lukutapahtuma.



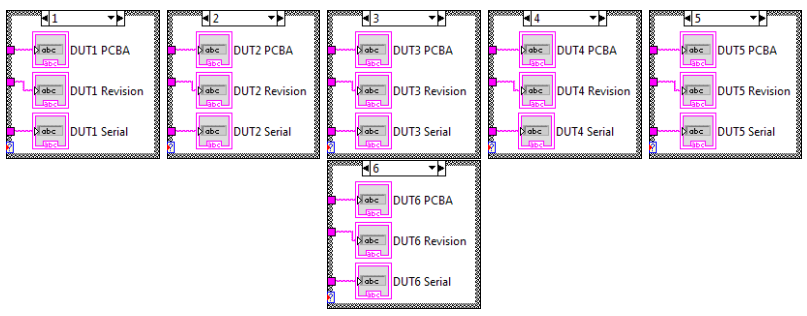
Kuva D.12: Tila 5. Revisiotila.



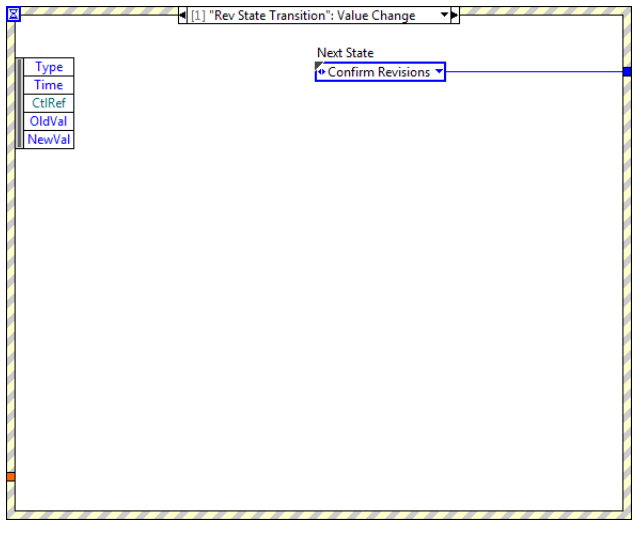
Kuva D.13: Käyttöliittymän väritys valitun latausaseman ohjelmointipaikan perusteella.



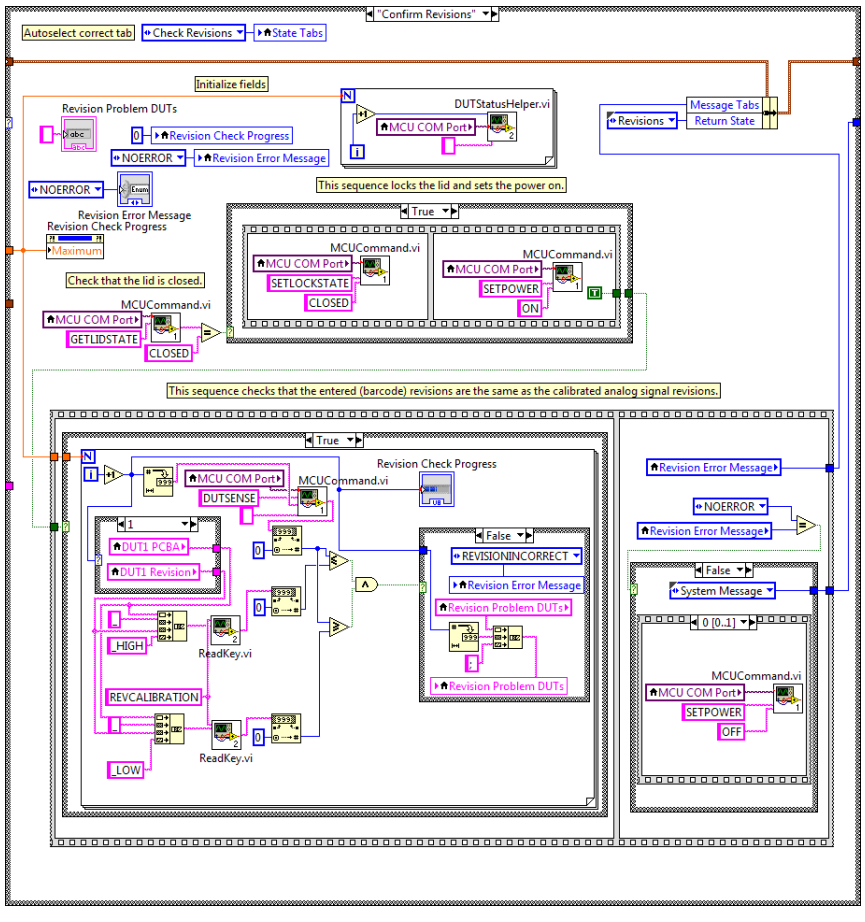
Kuva D.14: Vasemmalla ohjaus seuraavaan tilaan, kun käyttäjä on lukenut kaikkien ohjauskorttien viivakoodit. Oikealla olevassa ehtolauseen haarassa ei tehdä mitään, kun kansi on auki.



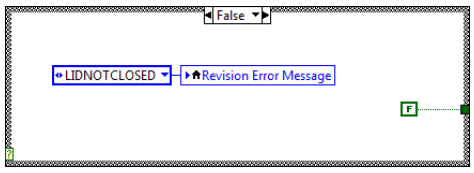
Kuva D.15: Ohjelmoitavan ohjauskortin nimen, revision ja sarjanumeron tallennus.



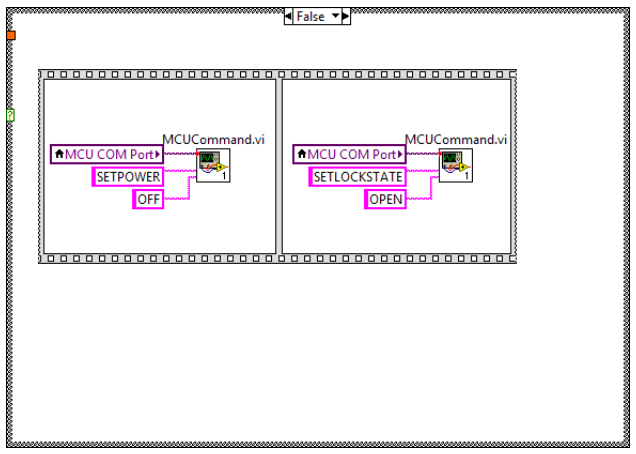
Kuva D.16: Revisoiden tarkistustilaan siirtyminen, kun aseman kansi suljetaan.



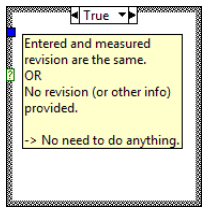
Kuva D.17: Tila 6. Syötettyjen revisioiden tarkistustila.



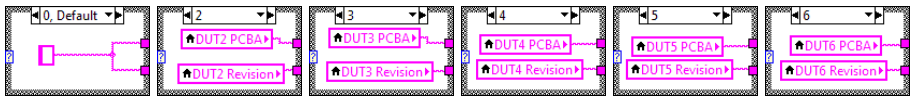
Kuva D.18: Jos latausaseman kansi on auki, ilmoitetaan siitä käyttäjälle.



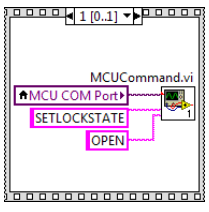
Kuva D.19: Jos kansi oli auki, sammutetaan aseman käyttöjännite ja avataan kannen lukko.



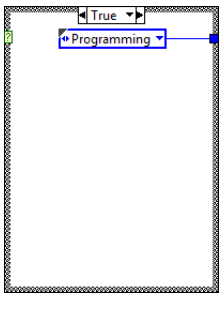
Kuva D.20: Jos revisioiden varmistustilassa esiintyi virhe, sammutetaan neulapedin käyttöjännite (kuva D.19) ja avataan lukko. Toisaalta jos revisioiden varmistustilassa ei ollut virheitä, siirrytään ohjelmointitilaan.



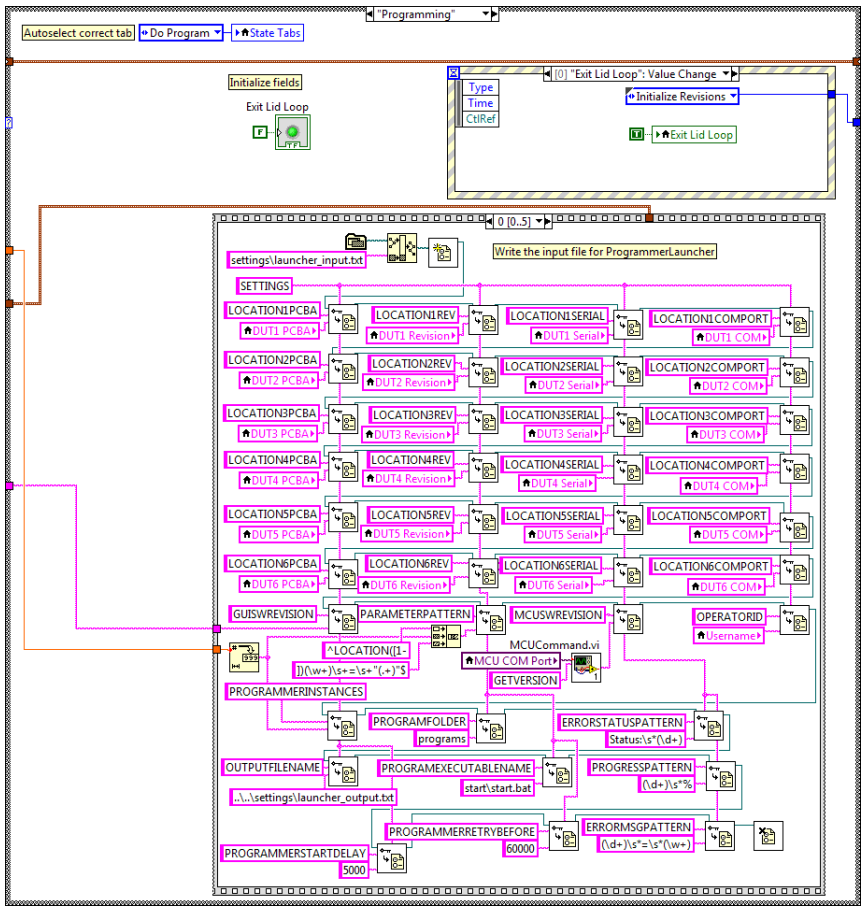
Kuva D.21: Valitaan käyttöliittymän tietokenttä, johon analogista revisiota verrataan.



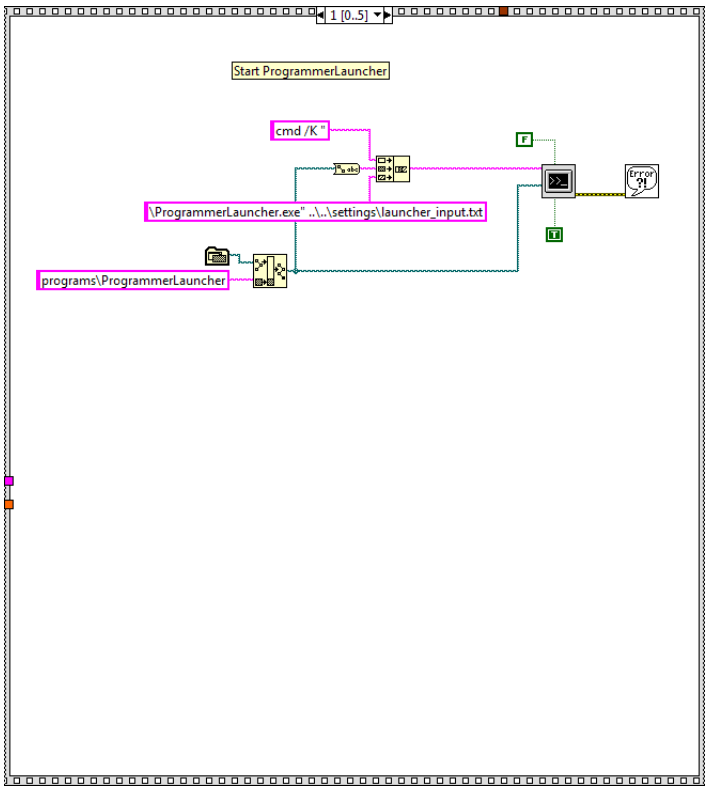
Kuva D.22: Jatkoa kuvan D.17 tilanteeseen, jossa revisioiden tarkastuksessa oli havaittu virhe: käyttöjännitteen sammuttamisen jälkeen kannen lukko avataan.



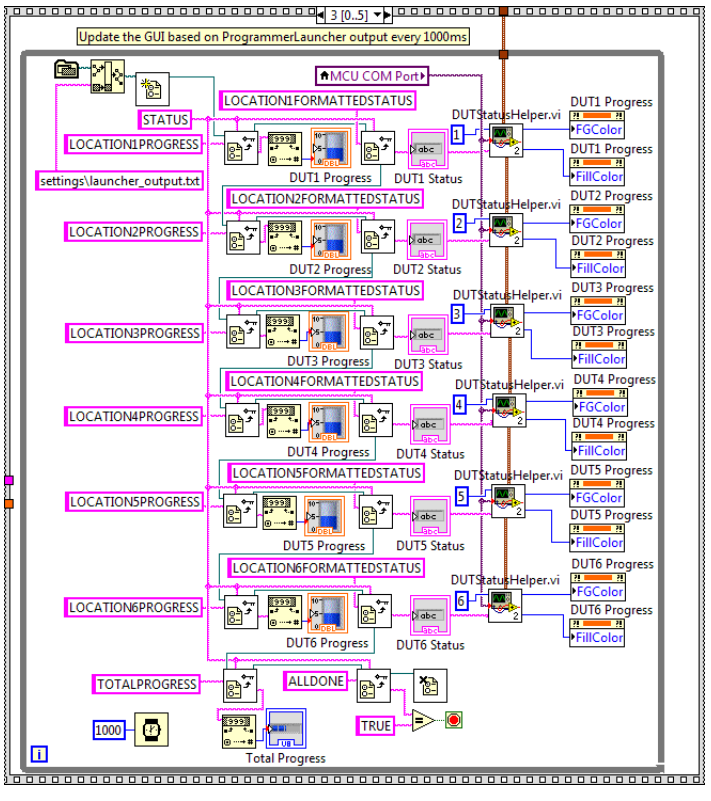
Kuva D.23: Jos kuvassa D.17 revisioiden tarkistuksessa ei ole virhettä, siirrytään vain seuraavaan tilaan.



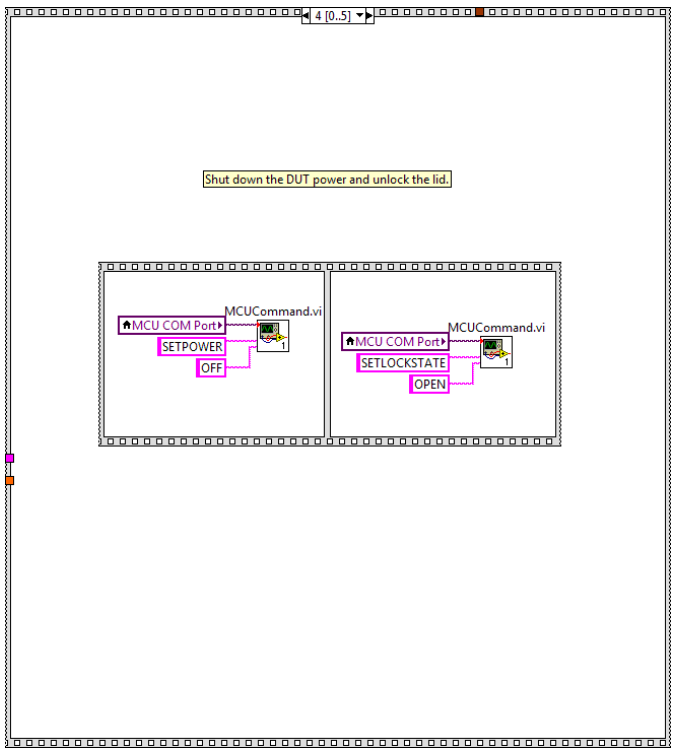
Kuva D.24: Tila 7. Ohjelmointitila.



Kuva D.25: Aloitetaan ohjauskorttien ohjelmointi ProgrammerLauncherilla.



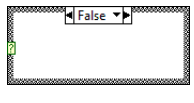
Kuva D.27: Luetaan ohjelmoinnin edistystieto tiedostosta sekunnin välein ja päivitetään käyttöliittymää.



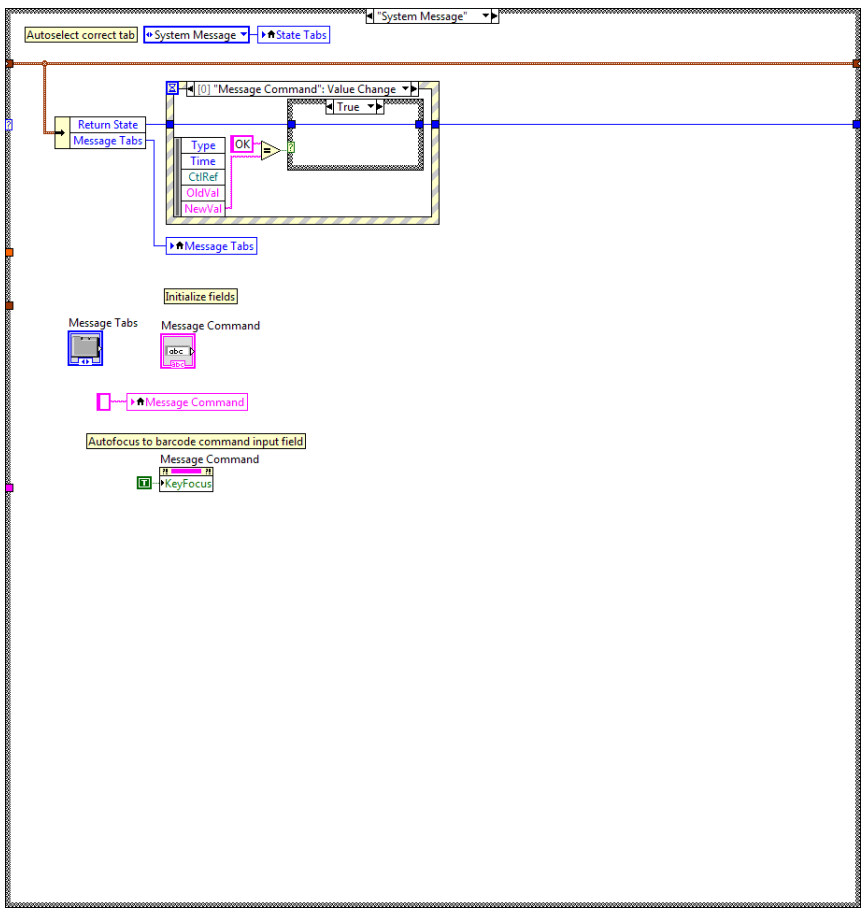
Kuva D.28: Lataus on valmis. Sammutetaan käyttöjännite ja avataan kannen lukko.

Kuva D.26: Odotetaan hetki, jotta ProgrammerLauncherilla on aikaa kirjoittaa uusi tilatieto tiedostoon. Muuten on vaarana, että luetaan vanha tiedosto, jossa lukisi, että ohjelmointi on loppunut.

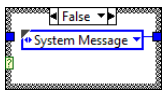
Kuva D.29: Jäädään odottamaan kannen avaamista.



Kuva D.30: Ei tehdä mitään, kun lataus on valmis ja kansi on kiinni.

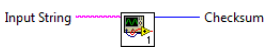


Kuva D.31: Tila 8. Ilmoitustila, jossa käyttäjälle ilmoitetaan virheestä.

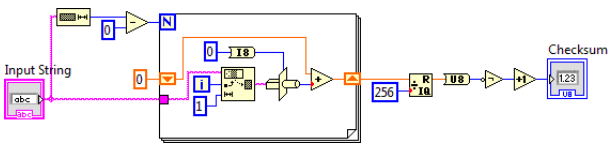


Kuva D.32: Jos ollaan luettu jokin muu viivakoodi kuin ”OK”, jolla palataan ilmoitustilasta takaisin, pysytään ilmoitustilassa.

Checksum.vi

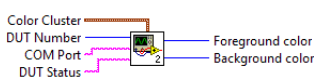


Kuva D.33: Tarkistussummalohkon symboli sisään- ja ulostuloineen.

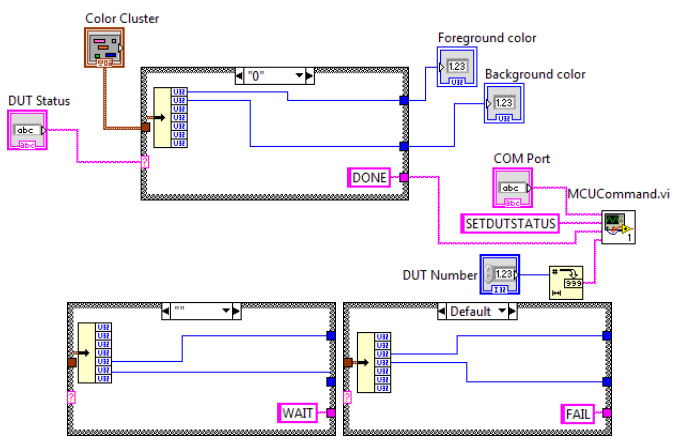


Kuva D.34: Kahdeksanbittisen kahden komplementti -tarkistussumman laskeminen syötteestä.

DUTStatusHelper.vi

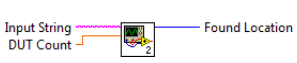


Kuva D.35: Ohjelmointipaikan värityksestä ja tilan kertovan valodiodin ohjauksesta vastaavan apulohkon symboli sisään- ja ulostuloineen.

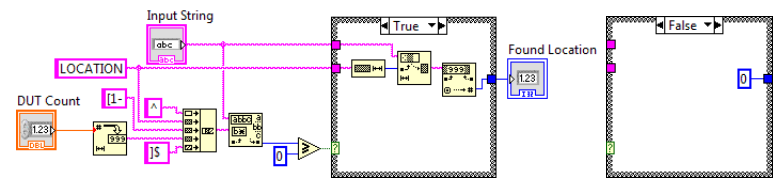


Kuva D.36: Ohjelmointipaikan väritys ja sen valodiodien ohjaus syötteen perusteella.

LocationFinder.vi



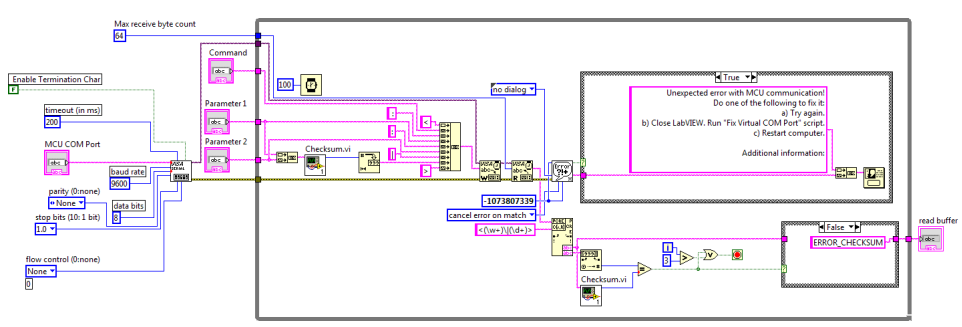
Kuva D.37: Viivakoodista ohjelmointipaikan parsivan apulohkon symboli sisään- ja ulostuloineen.



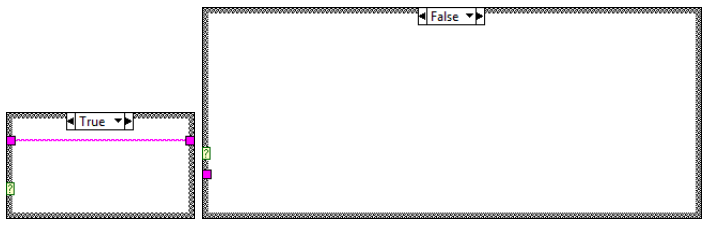
Kuva D.38: Apulohko palauttaa syötteessä olevan ohjelmointipaikan järjestysnumeron tai arvon nolla, jos syötteessä ei mainittu ohjelmointipaikkaa.

MCUCommand.vi

Kuva D.39: Mikrokontrollerille käskyn välittävän apulohkon symboli sisään- ja ulostuloineen.

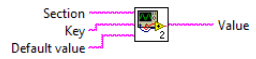


Kuva D.40: Apulohko välittää syötteenä saadun käskyn parametreineen mikrokontrollerille. Tiedon välitystä yritetään kolme kertaa automaattisesti. Jos mikrokontrolleri ei vastaa, annetaan käyttäjälle virheilmoitus. LabVIEW ei osaa lukea muuttuvamittaista viestiä sarjaportista, joten lohko aiheuttaa käytännössä aina aikakatkaisuvirheen (virhe -1073807339), jota ei näytetä käyttäjälle.

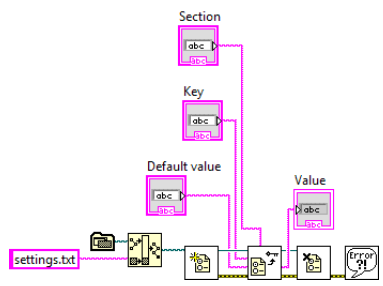


Kuva D.41: Jos käskyn välittämisessä ei ollut ongelmia, jatketaan tavallisesti.

ReadKey.vi

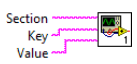


Kuva D.42: Asetusten lukemiseen käytetyn apulohkon symboli sisään- ja ulostuloineen.

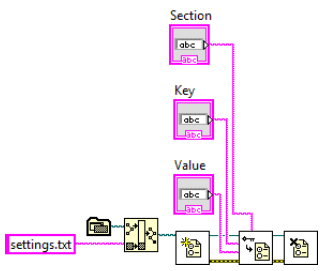


Kuva D.43: Arvon lukeminen asetustiedostosta avaimen perusteella.

WriteKey.vi



Kuva D.44: Asetusten kirjoittamiseen käytetyn apulohkon symboli sisään- ja ulostuloineen.



Kuva D.45: Arvon kirjoittaminen asetustiedostoon avaimen perusteella.

D.2 ProgrammerLauncher-ohjelman lähdekoodi

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.IO;
using System.Linq;
using System.Text;
using System.Text.RegularExpressions;
using System.Threading;
using System.Threading.Tasks;

namespace ProgrammerLauncher
{
  /// <summary>
  /// Launcher is the "main program" which launches multiple Programmers (subclass of Process) and monitors their progress. It is also responsible of writing handling the related files, including the progress log.
  /// </summary>
  public class ProgrammerLauncher
  {
    /// <summary>
    /// IMPORTANT! Versionnumber indicates the version of the Launcher (and Programmer). Version number is 1 when software is released to production and HAS TO BE INCREMENTED for all changes after that. Increment should be integer 1 (e.g. version numbers will be 2, 3, 4...).
    /// </summary>
    public readonly static int VERSIONNUMBER = 1;


    // Constants (read from settings file)
    private readonly string INPUTFILENAME;
    private readonly string OUTPUTFILENAME;
    private readonly int PROGRAMMERINSTANCES;
    private readonly string PROGRAMFOLDER;
    private readonly string PROGRAMEXECUTABLENAME;
    private readonly string PROGRESSPATTERN;
    private readonly string ERRORSTATUSPATTERN;
    private readonly string ERRORMSGPATTERN;
    private readonly string PARAMETERPATTERN;
    private readonly int PROGRAMMERSTARTDELAY;
    private readonly int PROGRAMMERRETRYBEFORE;

    // Static constants ('hard coded')
    private readonly static int OUTPUTWRITEINTERVAL = 1000; // in ms

    private Stopwatch _RunTimer;

    // Private variables
    private Programmer[] _Programmers;
    private List<string> _ErrorLog;

    // Log is used to format and write the log correctly.
    private Logger Log;


    /// <summary>
    /// Constructor of Launcher.
    /// </summary>
    /// <param name="InputFile">Name of the input file that includes all the settings required for operation. See more extensive documentation for the exact parameter names and expected values.</param>
    public ProgrammerLauncher(string InputFile)
    {
      INPUTFILENAME = InputFile;
      // Read the constants from the file before creating the Programmers
      string[] InputLines = new string[0];
      if (File.Exists(INPUTFILENAME))
      {
        InputLines = File.ReadAllLines(INPUTFILENAME);
        foreach (string Line in InputLines)
        {
          // The following "find value" operation expects that before and after the value there are no extra " characters
          int FirstQuote = Line.IndexOf("\"");
          int LastQuote = Line.LastIndexOf("\"");
          if (FirstQuote < 0 || LastQuote < 0 || FirstQuote == LastQuote)
          {
            continue;
          }
          string Value = Line.Substring(FirstQuote + 1, LastQuote - FirstQuote - 1).Replace("\\\\", "\\");

          switch (Line.Substring(0, Line.IndexOf("=")).Trim())
          {
            case "OUTPUTFILENAME":
              OUTPUTFILENAME = Value;
              break;
            case "PROGRAMMERINSTANCES":
              PROGRAMMERINSTANCES = Utility.ParseInt(Value);
              break;
            case "PROGRAMFOLDER":
              PROGRAMFOLDER = Value;
              break;
            case "PROGRAMEXECUTABLENAME":
              PROGRAMEXECUTABLENAME = Value;
              break;
            case "PROGRESSPATTERN":
              PROGRESSPATTERN = Value;
              break;
            case "ERRORSTATUSPATTERN":
              ERRORSTATUSPATTERN = Value;
              break;
            case "ERRORMSGPATTERN":
              ERRORMSGPATTERN = Value;
              break;
            case "PARAMETERPATTERN":
              PARAMETERPATTERN = Value;
              break;
            case "PROGRAMMERSTARTDELAY":
              PROGRAMMERSTARTDELAY = Utility.ParseInt(Value);
              break;
            case "PROGRAMMERRETRYBEFORE":
              PROGRAMMERRETRYBEFORE = Utility.ParseInt(Value);
              break;
            default:
              // Do nothing
              break;
          }
        }
      }


      // Now the constants are known and the Programmers can be initiated

      // Initialization

      _RunTimer = new Stopwatch();
      _RunTimer.Start();

      _Programmers = new Programmer[PROGRAMMERINSTANCES];
      for (int i = 0; i < PROGRAMMERINSTANCES; i++)
      {
        _Programmers[i] = new Programmer(this);
      }

      _ErrorLog = new List<string>();


      // Read PCBA names and revisions from input file
      foreach (string Line in InputLines)
      {
        int Location = 0;
        string Key = "";
        string Value = "";

        // For clarity: rexpattern looks for string of the following syntax 'LOCATION[index][attribute name] = "[non-empty attribute value]"'
        Match Match = Regex.Match(Line, PARAMETERPATTERN);
        if (Match.Success)
        {
          Location = Utility.ParseInt(Match.Groups[1].Value) - 1; // -1 because the array index is 0-based
          Key = Match.Groups[2].Value;
          Value = Match.Groups[3].Value;
        }
        else
        {
          // No match, skip to next line
          continue;
        }


        switch (Key)
```

```
151         {
152           case "PCBA":
153             _Programmers[Location].Pcba = Value;
154             break;
155           case "REV":
156             _Programmers[Location].Rev = Value;
157             break;
158           case "COMPORT":
159             _Programmers[Location].Comport = Value;
160             break;
161           default:
162             // Do nothing
163             break;
164         }
165       }
166
167       // Start logging
168       Log = new Logger();
169       Log.StartLogging(INPUTFILENAME);
170     }
171
172
173     /// <summary>
174     /// LaunchProgrammers is used to start the initiated Programmer processes. If no Programmers have been initiated, this function does nothing.
175     /// </summary>
176     public void LaunchProgrammers()
177     {
178       foreach (Programmer Prog in _Programmers)
179       {
180         Prog.StartProgramming();
181       }
182     }
183
184
185     /// <summary>
186     /// IsRunning checks whether ANY Programmer is still running.
187     /// </summary>
188     /// <returns>True if ANY Programmer is running, False if all Programmers have either finished or failed.</returns>
189     public bool IsRunning()
190     {
191       foreach (Programmer Prog in _Programmers)
192       {
193         if (Prog.IsRunning)
194         {
195           return true;
196         }
197       }
198       return false;
199     }
200
201
202     /// <summary>
203     /// LogErrorMessage logs an error which will later be appended to the log which will be written.
204     /// </summary>
205     /// <param name="Error">Error message to be logged.</param>
206     public void LogErrorMessage(string Error)
207     {
208       _ErrorLog.Add(Error);
209     }
210
211
212     /// <summary>
213     /// WriteOutput writes the output file which includes statuses and errors of different Programmer instances. It also writes the error log (if any, see function LogErrorMessage).
214     /// </summary>
215     public void WriteOutput(bool BeforeLaunching = false)
216     {
217       string Contents = "[STATUS]\r\n";
218
219       bool IsDone = true;
220
221       // Write the state of programmer
222       for (int i = 0; i < PROGRAMMERINSTANCES; i++)
223       {
224         string ErrorStatus = _Programmers[i].ErrorStatus == -1 ? "" : ""+ _Programmers[i].ErrorStatus;
225         Contents += "LOCATION" + (i + 1) + "PROGRESS = \"" + _Programmers[i].Progress + "\"\r\n";
226         Contents += "LOCATION" + (i + 1) + "ERRORSTATUS = \"" + ErrorStatus + "\"\r\n";
227         Contents += "LOCATION" + (i + 1) + "FORMATTEDSTATUS = \"" + _Programmers[i].FormattedStatus + "\"\r\n";
228
229         // Errorstatus -1 is default value and will be changed when success or error occurs
230         if (_Programmers[i].IsRunning && _Programmers[i].ErrorStatus < 0)
231         {
232           IsDone = false;
233         }
234         else
235         {
236           // End logging for this programmer. This may be called multiple times. Logger ignores all but the first call for [i]th Programmer.
237           Log.NotifyProgrammerEnded(i, _Programmers[i].ErrorStatus);
238         }
239       }
240
241       // If all programmers have finished (error or success), IsDone is true
242       // Except if BeforeLaunching is true, it means that the GUI should wait for the actual programming to begin. In this case, write FALSE.
243       Contents += "ALLDONE = \"" + (IsDone && !BeforeLaunching ? "TRUE" : "FALSE") + "\"\r\n";
244       if (IsDone)
245       {
246         // Write the log once all Programmers have finished. If this is called multiple times, Logger ignores all but the first call.
247         Log.EndLogging();
248       }
249
250       // Calculate total progress
251       int PcbasToProgram = 0;
252       int ProgressSum = 0;
253       foreach (Programmer Prog in _Programmers)
254       {
255         if (Prog.Rev != "")
256         {
257           PcbasToProgram++;
258           ProgressSum += Prog.Progress;
259         }
260       }
261       int TotalProgress = (PcbasToProgram == 0 ? 100 : ProgressSum / PcbasToProgram);
262       // Before the launching of the programmers, the progress is 0.
263       if (BeforeLaunching) TotalProgress = 0;
264       Contents += "TOTALPROGRESS = \"" + TotalProgress + "\"\r\n";
265
266
267       // Write error log
268       Contents += "ERRORS = \"";
269       foreach (string Error in _ErrorLog)
270       {
271         Contents += Error + "\r\n";
272       }
273       Contents += "\"";
274
275
276       // Output the file
277       File.WriteAllText(OUTPUTFILENAME, Contents);
278     }
279
280
281     /// <summary>
282     /// The Main function of the program.
283     /// </summary>
284     /// <param name="Args">There are two possible arguments of which only one should be used at a time. If the parameter is VERSION (ignore case), the program outputs version and exits. In any other case the argument is interpreted as a file name (and relative path if in different directory) of the input file which contains the settings for the Launcher.</param>
285     static void Main(string[] Args)
286     {
287
288       // DEBUG!!!
289
290       FileStream filestream = new FileStream("out.txt", FileMode.Create);
291       var streamwriter = new StreamWriter(filestream);
292       streamwriter.AutoFlush = true;
293       Console.SetOut(streamwriter);
294       Console.SetError(streamwriter);
```

```

    // DEBUG ENDS!!!

    if (Args.Length < 1)
    {
      Console.WriteLine("Too few arguments! You have to specify the input file name (relative to the current directory). Or alternatively \"VERSION\" (ignore case) to get the version.");
      return;
    }

    // If the argument and the only argument is version (ignore case), print the version number and exit.
    if (Args.Length == 1 && Args[0].ToLower() == "version")
    {
      Console.WriteLine(VERSIONNUMBER);
      return;
    }

    Console.WriteLine("Starting to launch Programmer processes.");

    // Initialization
    ProgrammerLauncher Launch = new ProgrammerLauncher(String.Join(" ", Args[0]));

    // Write empty status file before launching the programmers, because it will take time.
    try
    {
      Launch.WriteOutput(true);
    }
    catch (IOException Ioe)
    {
      // The usual/probable error is that the file is locked. Log the error and exit. Otherwise there may be further problems (such as the GUI will use the old output file instead of new one).
      Console.WriteLine(Ioe.ToString());
      return;
    }

    // Start programming
    Launch.LaunchProgrammers();

    Console.WriteLine("Programmers are running.");

    // Monitor programming status and write status to temporary file
    while (Launch.IsRunning())
    {
      try
      {
        Launch.WriteOutput();
      }
      catch (IOException Ioe)
      {
        // The usual/probable error is that the file is locked. No need to do anything in that case, but log the error in any case.
        Console.WriteLine(Ioe.ToString());
      }

      // Sleep a small while so that the host computer does not get overwhelmed.
      Thread.Sleep(OUTPUTWRITEINTERVAL);
    }


    // Now all programmers have finished. write the output (and be sure to do that even if the file may be locked every now and then!)
    bool ProgressWritten = false;
    while (!ProgressWritten)
    {
      try
      {
        Launch.WriteOutput();
        ProgressWritten = true;
      }
      catch (IOException ioe)
      {
        Console.WriteLine(ioe.ToString());

        // Sleep a small while so that the host computer does not get overwhelmed.
        Thread.Sleep(OUTPUTWRITEINTERVAL);
      }
    }


    // Exit when all programmers have finished

    Console.WriteLine("All Programmers have finished. Press Enter to exit.");
    //Console.ReadLine(); // USE ONLY WHEN DEBUGGING! OTHERWISE PROCESSES WILL BE LEFT HANGING!
  }


  /// <summary>
  /// Programmer is an individual Process (subclass of it) which writes the actual software to Cypress PCBAs. One Programmer per Cypress is used and these can be executed in parallel.
  /// </summary>
  public class Programmer
  {
    private string _Pcba;
    private string _Rev;
    private string _Comport;
    private int _Progress; // -1 is failure (or default), 0-100 is progress
    private int _ErrorStatus; // -1 is default, 0 is success, > 0 is error
    private string _ErrorMessage; // Error message if available
    private bool _IsRunning = false;
    private bool _HasRetried = false; // In case of failure, if and time elapsed is less than PROGRAMMERRETRYBEFORE, retry programming once.

    // for the following two variables: -1 is uninitialized, 0 is default (just "start.bat"), > 0 is "start1.bat", "start2.bat", etc.
    private int _ExecutableCount = -1; // number of executables to be run.
    private int _CurExecutable = -1; // current index of executables to run.

    private Process ProgrammerProcess;

    // Reference to the parent Launcher
    private ProgrammerLauncher _Parent;


    private readonly string LabViewDirectory = Directory.GetParent(Directory.GetParent( Directory.GetCurrentDirectory() ).ToString()).ToString();

    /// <summary>
    /// Get and set Pcba property. This should be in the material code format (3AUA... or 3AXD...). REQUIRED property for Programmer.
    /// </summary>
    public string Pcba
    {
      set { this._Pcba = value; }
      get { return this._Pcba; }
    }

    /// <summary>
    /// Get and set Rev property. This should be in the letter format (as many letters as are required). REQUIRED property for Programmer.
    /// </summary>
    public string Rev
    {
      set { this._Rev = value; }
      get { return this._Rev; }
    }

    /// <summary>
    /// Get and set Comport property. This should be in format of COM[numbers]. E.g. COM1 or COM34. REQUIRED property for Programmer.
    /// </summary>
    public string Comport
    {
      set { this._Comport = value; }
      get { return this._Comport; }
    }

    /// <summary>
    /// Get progress of this Programmer. Progress is from 0-100 where 100 means finished. -1 if an error has occurred.
    /// </summary>
    public int Progress
```

```
    {
      get { return _Progress; }
    }

    /// <summary>
    /// Get error status. > 0 in case of failure and 0 if successful. -1 if neither (default value).
    /// </summary>
    public int ErrorStatus
    {
      // for error status -2 (waiting for another executable to run), show -1 for GUI compatibility
      get { return (_ErrorStatus < 0 ? -1 : _ErrorStatus); }
    }

    /// <summary>
    /// PASS or FAIL + ErrorMessage (or ErrorStatus if message is not available)
    /// </summary>
    public string FormattedStatus
    {
      get {
        string FormattedStr = "";
        // If status is incomplete, return empty string
        if (ErrorStatus < 0)
        {
          return FormattedStr;
        }
        // Otherwise return the correct PASS or FAIL and status message
        else
        {
          FormattedStr = (_ErrorStatus == 0 ? "PASS" : "FAIL");
          // Only include message when FAIL
          if (_ErrorStatus > 0)
          {
            FormattedStr += " (" + (_ErrorMessage == "" ? "" + _ErrorStatus : _ErrorMessage) + ")";
          }
          return FormattedStr;
        }
      }
    }

    /// <summary>
    /// Returns true if there is an executable that is still running or will be run.
    /// </summary>
    public bool IsRunning
    {
      get { return this._IsRunning; }
    }

    /// <summary>
    /// Sets HasRetried or gets it. True if the executable has failed and retried and false if it has not retried. This may be because it has not failed or the time limit for retry has expired.
    /// </summary>
    public bool HasRetried
    {
      set { this._HasRetried = value; }
      get { return this._HasRetried; }
    }


    /// <summary>
    /// Constructor of Programmer.
    /// </summary>
    /// <param name="Parent">Referense to the parent Launcher.</param>
    public Programmer(ProgrammerLauncher Parent) : base()
    {
      _Parent = Parent;

      _Pcba = "";
      _Rev = "";
      _Comport = "";
      _Progress = -1;
      _ErrorStatus = -1;
      _ErrorMessage = "";
    }


    /// <summary>
    /// Finds out how many executables should be executed. Then calls another function which executes the first executable.
    /// </summary>
    /// <returns>True if launching was successful, False if unsuccessful (check that the required properties have been set).</returns>
    public bool StartProgramming()
    {
      // If important information is missing, do not do anything more.
      if (_Pcba == "" || _Rev == "" || _Comport == "")
      {
        return false;
      }


      // Find the correct number of executables in the target folder
      string ExecutableDirectory = LabViewDirectory + Path.DirectorySeparatorChar + _Parent.PROGRAMFOLDER + Path.DirectorySeparatorChar + _Pcba + Path.DirectorySeparatorChar + _Rev + Path.DirectorySeparatorChar + _Parent.PROGRAMEXECUTABLENAME.Substring(0, _Parent.PROGRAMEXECUTABLENAME.LastIndexOf( Path.DirectorySeparatorChar));
      List<string> ExecutableCandidates = new List<string>(Directory.GetFiles(ExecutableDirectory));

      // Step 1: look for default "start.bat". if found, no need to look further
      foreach (string ExecutableCandidate in ExecutableCandidates)
      {
        if (ExecutableCandidate.EndsWith( _Parent.PROGRAMEXECUTABLENAME))
        {
          _ExecutableCount = 0;
          _CurExecutable = 0;
          break;
        }
      }

      // Step 2: if default "start.bat" was not found, look for "start1.bat", "start2.bat", etc.
      if (_ExecutableCount < 0)
      {
        string ExecutableNameRegex = "";
        if (_Parent.PROGRAMEXECUTABLENAME.Contains( Path.DirectorySeparatorChar))
        {
          int StartLoc = _Parent.PROGRAMEXECUTABLENAME.LastIndexOf( Path.DirectorySeparatorChar) + 1;
          int Len = _Parent.PROGRAMEXECUTABLENAME.LastIndexOf(".") - StartLoc;
          ExecutableNameRegex += _Parent.PROGRAMEXECUTABLENAME.Substring(StartLoc, Len);
        }
        else
        {
          ExecutableNameRegex += _Parent.PROGRAMEXECUTABLENAME.Substring(0, _Parent.PROGRAMEXECUTABLENAME.LastIndexOf("."));
        }
        ExecutableNameRegex += "[1-9]";
        ExecutableNameRegex += _Parent.PROGRAMEXECUTABLENAME.Substring( _Parent.PROGRAMEXECUTABLENAME.LastIndexOf(".")) + "$";

        foreach (string ExecutableCandidate in ExecutableCandidates)
        {
          if (Regex.IsMatch(ExecutableCandidate.Substring( ExecutableCandidate.LastIndexOf( Path.DirectorySeparatorChar) + 1), ExecutableNameRegex))
          {
            _ExecutableCount = (_ExecutableCount < 0 ? 1 : _ExecutableCount + 1);
            _CurExecutable = 1;
          }
        }
      }

      Console.WriteLine("Programmer at COM port "+ _Comport +" has "+ _ExecutableCount +" executables to execute. 0 means default (not
```

```
↩ numbered) executable.");
564
565         // Step 3: if there was no "start[n].bat" ↩
↩ files, fail. Otherwise, do program.
566         if (_CurExecutable < 0)
567         {
568           return false;
569         }
570         else
571         {
572           return ↩
↩ StartProgrammingExecutable(_CurExecutable);
573         }
574       }
575
576
577
578       /// <summary>
579       /// Starts programming IF the following ↩
↩ properties have been set: Pcba, Rev and Comport.
580       /// </summary>
581       /// <param name="ExecutableNumber">Number of ↩
↩ executable in case of multiple ↩
↩ executables.</param>
582       /// <returns>True if the starting was ↩
↩ successful.</returns>
583       private bool StartProgrammingExecutable(int ↩
↩ ExecutableNumber)
584       {
585         // Check that all the necessary information is ↩
↩ known, and if so, start programming.
586         if (_Pcba != "" && _Rev != "" && _Comport != "")
587         {
588           Console.WriteLine("Programmer at COM port " ↩
↩ + _Comport + " is starting executable number " + ↩
↩ ExecutableNumber + ".");
589
590           string ExecutableFileNameBegin = ↩
↩ _Parent.PROGRAMEXECUTABLENAME.Substring(0, ↩
↩ _Parent.PROGRAMEXECUTABLENAME.LastIndexOf("."));
591           string ExecutableFileNameEnd = ↩
↩ _Parent.PROGRAMEXECUTABLENAME.Substring( ↩
↩ _Parent.PROGRAMEXECUTABLENAME.LastIndexOf("."));
592
593           string ExecutableFileNameModifier = "";
594           if (ExecutableNumber > 0)
595           {
596             ExecutableFileNameModifier = ""+ ↩
↩ ExecutableNumber;
597           }
598
599           ProgrammerProcess = new Process();
600
601           ProgrammerProcess.StartInfo.FileName = ↩
↩ LabViewDirectory + Path.DirectorySeparatorChar + ↩
↩ _Parent.PROGRAMFOLDER + ↩
↩ Path.DirectorySeparatorChar + _Pcba + ↩
↩ Path.DirectorySeparatorChar + _Rev + ↩
↩ Path.DirectorySeparatorChar + ↩
↩ ExecutableFileNameBegin + ↩
↩ ExecutableFileNameModifier + ↩
↩ ExecutableFileNameEnd;
602           ProgrammerProcess.StartInfo.WorkingDirectory ↩
↩ = LabViewDirectory + Path.DirectorySeparatorChar ↩
↩ + _Parent.PROGRAMFOLDER + ↩
↩ Path.DirectorySeparatorChar + _Pcba + ↩
↩ Path.DirectorySeparatorChar + _Rev + ↩
↩ Path.DirectorySeparatorChar + ↩
↩ ExecutableFileNameBegin.Substring(0, ↩
↩ ExecutableFileNameBegin.LastIndexOf( ↩
↩ Path.DirectorySeparatorChar));
603           ProgrammerProcess.StartInfo.Arguments = ↩
↩ _Comport;
604           ProgrammerProcess.StartInfo.UseShellExecute ↩
↩ = false;
605           ↩
↩ ProgrammerProcess.StartInfo.RedirectStandardOutput ↩
↩ = true;
606
607           ProgrammerProcess.EnableRaisingEvents = true;
608           ProgrammerProcess.OutputDataReceived += new ↩
↩ DataReceivedEventHandler(UpdateProgress);
609           // The Process.Exited event SHALL NOT BE ↩
↩ USED! It is triggered before all data has ↩
↩ arrived which may result in incorrect behaviour.
610           //ProgrammerProcess.Exited += new ↩
↩ EventHandler(ExecutableExited);
611
612           Console.WriteLine(_Comport + " is starting ↩
↩ program " + ProgrammerProcess.StartInfo.FileName ↩
↩ + " at " + ↩
↩ ProgrammerProcess.StartInfo.WorkingDirectory);
613
614           _IsRunning = true;
615
616           // Wait few seconds before starting the ↩
↩ executable. There are three reasons (and cases!) ↩
↩ when the delay is necessary.
617           // 1) When the DUT is powered on, it will ↩
↩ take small while to be ready for programming
618           // 2) If programmers are launched exactly at ↩
↩ the same time, there will be error (presumably ↩
↩ because they try to access the same resources)
619           // 3) When there are multiple software to be ↩
↩ uploaded, if the next is started instantly after ↩
↩ the previous, there will be a connection error
620           // The delay is implemented as ↩
↩ 'in-pieces-sleep' so that the status file is ↩
↩ generated in the background.
621           Stopwatch timer = new Stopwatch();
622           timer.Start();
623           while (timer.ElapsedMilliseconds < ↩
↩ _Parent.PROGRAMMERSTARTDELAY)
624           {
625             _Parent.WriteOutput();
626             Thread.Sleep(OUTPUTWRITEINTERVAL);
627           }
628           timer.Stop();
629
630           ProgrammerProcess.Start();
631           ProgrammerProcess.BeginOutputReadLine();
632
633           return true;
634         }
635
636         // Default return
637         return false;
638       }
639
640
641       /// <summary>
642       /// Event handler for Programmer exit. If there ↩
↩ are more executables waiting and the exited one ↩
↩ was successful, launch the next one after a ↩
↩ small delay.
643       /// </summary>
644       private void ExecutableExited()
645       {
646         Console.WriteLine("Exit event at COM port " + ↩
↩ _Comport + ".");
647
648         // If errorstatus = 0 (success) stop running. ↩
↩ loading was success.
649         if (_ErrorStatus == 0)
650         {
651           Console.WriteLine("Programmer at COM port "+ ↩
↩ _Comport +" exited successfully.");
652           _Progress = 100;
653           _IsRunning = false;
654         }
655         // If errorstatus = -2, there are more ↩
↩ executables to be run
656         else if (_ErrorStatus == -2)
657         {
658           _CurExecutable++;
659
660           StartProgrammingExecutable(_CurExecutable);
661         }
662         // If there was an error, stop programming and ↩
↩ set progress to finished
663         else
664         {
665           if (!HasRetried && ↩
↩ _Parent._RunTimer.ElapsedMilliseconds < ↩
↩ _Parent.PROGRAMMERRETRYBEFORE)
666           {
667             Console.WriteLine("Programmer at COM port ↩
↩ " + _Comport + " is retrying at time instance " ↩
↩ + _Parent._RunTimer.ElapsedMilliseconds + " ms.");
668             HasRetried = true;
669             StartProgrammingExecutable(_CurExecutable);
670           }
671           else
672           {
673             Console.WriteLine("Programmer at COM port ↩
↩ " + _Comport + " failed and exited.");
674             _Progress = 100;
675             _IsRunning = false;
676           }
677         }
678       }
679
680
681       /// <summary>
```

```
    /// UpdateProgress updates the progress of this Programmer by parsing the command line programs output and looking for the patterns set in the settings file (input file passed to the Main of Launcher). This function is called automatically by .NET when there is new text in the command line.
    /// </summary>
    /// <param name="SendingProcess">.NET's default parameter. It is not used.</param>
    /// <param name="OutLine">Output of the software loading process.</param>
    private void UpdateProgress(object SendingProcess, DataReceivedEventArgs OutLine)
    {

      // When OutLine.Data is null, it means that the process has exited
      if (OutLine.Data == null)
      {
        ExecutableExited();
      }

      // Ignore line if it is empty
      if (String.IsNullOrEmpty(OutLine.Data))
      {
        return;
      }

      // debug Console.WriteLine(_Comport +":> "+ OutLine.Data);

      // Get progress by finding any percent.
      // The original pattern was formatted to catch the following type of messages: " 0 % downloaded /". (or more simply: "[number][optional white spaces]%")
      Match ProgressMatch = Regex.Match(OutLine.Data, _Parent.PROGRESSPATTERN);
      if (ProgressMatch.Success)
      {
        _Progress = Utility.ParseInt(ProgressMatch.Groups[1].Value);

        // if there are multiple executables, modify the progress accordingly
        if (_ExecutableCount > 0)
        {
          _Progress = (100 * (_CurExecutable-1) + _Progress) / _ExecutableCount;
        }

        Console.WriteLine("At " + _Comport + " progress is " + _Progress + " %.");
      }

      // In case of failure, get the error code.
      // The original string was "Status: 38", or simply "Status:[optional white spaces][status number]"
      Match ErrorStatusMatch = Regex.Match(OutLine.Data, _Parent.ERRORSTATUSPATTERN);
      if (ErrorStatusMatch.Success)
      {
        int NewErrorStatus = Utility.ParseInt(ErrorStatusMatch.Groups[1].Value);

        // If error occurs, progress is reset to indicate failure EXCEPT if the error = 0 which means success
        if (NewErrorStatus != 0)
        {
          _ErrorStatus = NewErrorStatus;
          _Progress = -1;
        }
        else
        {

          // if there are still executables to run, set errorstatus to -2 to signal continue. Otherwise to 0.
          if (_ExecutableCount != _CurExecutable)
          {
            _ErrorStatus = -2;
          }
          else
          {
            _ErrorStatus = NewErrorStatus;
            _Progress = 100;
          }
        }
      }

      // Once the error status is known, wait for the error message list and find the value for the code from the list.

      Match ErrorMessageMatch = Regex.Match(OutLine.Data, _Parent.ERRORMSGPATTERN);
      if (ErrorMessageMatch.Success)
      {
        if (ErrorMessageMatch.Groups[1].Value == ""+ _ErrorStatus)
        {
          _ErrorMessage = ErrorMessageMatch.Groups[2].Value;
        }
      }

    }
}


/// <summary>
/// Utility functions which are convenient regardless of the class.
/// </summary>
public class Utility
{
  /// <summary>
  /// Parses String into Integer. Unlike the .NET's int.Parse function, if this function cannot parse the value, returns 0 instead of failing.
  /// </summary>
  /// <param name="Str"></param>
  /// <returns></returns>
  public static int ParseInt(string Str)
  {
    int Result;
    return int.TryParse(Str, out Result) ? Result : 0;
  }
}
```