PIIRIKORTTISARJAN TESTAUSJÄRJESTELMÄ

Ilari Virta
Opinnäytetyö
3.7.2010
Tietotekniikan koulutusohjelma
Oulun seudun ammattikorkeakoulu

OULUN SEUDUN AMMATTIKORKEAKOULU TIIVISTELMÄ

Koulutusohjelma	Opinnäytetyö	Sivuja + Liitteitä
Tietotekniikka	Insinöörityö	50+0
Suuntautumisvaihtoehto	Aika	
Elektroniikan suunnittelu –ja testaus	3.7.2010	
Työn tilaaja	Työn tekijä	
Xemec OY	Ilari Virta	
Työn nimi		
Piirikorttisarjan testausjärjestelmä		
Avainsanat		
Mikrokontrolleri, C-kieli, Python, I/O, CAN, Linux, Windows		

Opinnäytetyön aiheena oli piirikorttisarjan testausjärjestelmän suunnittelu. Testausjärjestelmä tehtiin ensi sijassa Xemec OY:n käyttöön, mutta tulevaisuudessa se tullaan luovuttamaan myös piirikorttien kokoonpanijan käyttöön. Testausjärjestelmän tarkoitus on nopeuttaa sekä helpottaa piirikorttien toiminnallista testausta sekä havaitun vian paikantamista. Opinnäytetyö toteutettiin projektimuotoisena työnä keväällä 2010. Projekti sisälsi järjestelmän vaatimusten määrittelyn, ohjelmiston ja elektroniikan suunnittelun, toteutuksen sekä testauksen.

Testausjärjestelmä sisältää piirikorttien mikrokontrollereille ladattavan testiohjelman, testaukseen vaadittavan lisäelektroniikan sekä käyttöliittymäsovelluksen PC:lle. Järjestelmän suunnittelussa pyrittiin hyödyntämään mahdollisimman paljon piirikorttien omaa elektroniikkaa, jotta testauksen vaatima lisäelektroniikka saataisiin mahdollisimman yksinkertaiseksi ja luotettavaksi. Piirikorttien mikrokontrollereille ladattava testiohjelma tehtiin mahdollisimman yksinkertaiseksi ja nopeaksi, jolloin pienen laskentatehon omaavien mikrokontrollerien toiminta ei hidasta testin suorittamista. Suurin osa ohjelmiston toiminnasta pyrittiin toteuttamaan käyttöliittymäsovelluksessa. Ohjelmiston kirjoittamiseen käytettiin sekä C-, että Python-ohjelmointikieliä. Testauksen vaatima lisäelektroniikka sisältää pääosin passiivisia diskreettikomponentteja. Piirilevyn prototyyppi rakennettiin käsin juotettavista aksiaalikomponenteista, mutta tulevaisuudessa piirilevy tullaan suunnittelemaan ja valmistuttamaan myös pintaliitoskomponenteille, jolloin lisäelektroniikka saadaan integroitua mahdollisimman pieneen tilaan.

Projektin lopputuloksena syntyi testausjärjestelmä, joka testaa piirikorttisarjan korttien I/O-liitännät, paikantaa testissä ilmenneet viat sekä tallentaa testin tulokset tekstitiedostoon. Käyttöliittymäsovellus on sovitettu sekä Linux-, että Windows-käyttöjärjestelmille.

OULU UNIVERSITY OF APPLIED SCIENCES ABSTRACT

Degree programme	Thesis	Number of pages + appendices
Information Technology and Telecommunications	B.Sc.	50+0
Line	Date	
Electronics Design and Testing	3.7.2010	
Commissioned by	Author	
Xemec OY	Ilari Virta	

Thesis title
PWB Testing System

Keywords
Microcontroller, C-Language, Python, I/O, CAN, Linux, Windows

The title of this thesis is a PWB Testing System. The testing system is done primarily for Xemec LTD's service, but will be given for the use of circuit boards assembler in future. The testing system is intended to expedite and facilitate functional testing of PWBs as well as finding and locating faults. Thesis was carried out as a project work in the spring of 2010. The project included a system requirements definition, software and electronics design, implementation and testing.

The testing system includes a microcontroller downloadable test program, required additional electronics and user interface application for PC. The system has been designed so that it uses PWBs own electronics as much as possible to make the additional electronics as simple and reliable as possible. The test software have been made as simple as possible and fast, when a small computational power of its own microcontroller action does not slow down the test. Most of the activities of the software has tried to implement to the user interface application.
Software is written in C and Python programming languages. Testing electronics contains mostly discrete passive-components. Testing electronics prototype has been built by hand-solder components, but will be designed and manufactured for surface-mount components in future to get the PWB integrated in a smallest possible space.

The result of this project is a working testing system, that tests PWBs I/O-connections, locates faults and saves test results to a text file. The interface application is ported for Linux, and Windows operating systems.

SISÄLTÖ

SANASTO

Aksiaalikomponentti	Jalallinen komponentti
CAN	Controller Area Network, väylätyyppi
Debuggaus	Virheen etsintä
Diskreettikomponentti	Yksittäinen komponentti
Event	Tapahtuma
Flash	Muistityyppi
GNU	Vapaita ohjelmistoja tuottava projekti
GPL	General Public Licence, lisenssi-tyyppi
I/O	Input/Output, sisäänmeno/ulostulo
Image	ROM-muistiin ladattava konekielinen tiedosto
Importata	Liittää eri tiedostossa olevaa koodia lähdekoodiin
LED	Light Emitting Diode, valodiodi
LIN	Local Interconnect Network, väylätyyppi
Pollata	Tutkia aktiivisesti muuttujan tilaa
PWB	Printed Wiring Board, Painopiirilevy
RX-buffer	Saapuvan tiedon puskurimuisti
Spice	Simulation Program with Integrated Circuit Emphasis, Piirikaavion simuloinnissa käytettävä simulointimalli
Triggaus	Tapahtuman liipaisu

1 JOHDANTO

Opinnäytetyöprojektissa kehitettiin Xemec OY:n valmistaman automaatiolaitteen piirikorttisarjan testausjärjestelmä. Opinnäytetyön aiheen taustalla oli tilaajan tarve piirikorttisarjan testausta helpottavalle järjestelmälle, sillä aikaisemmin piirikortit on testattu toimivassa järjestelmässä yksi kerrallaan, jolloin aikaa on kulunut huomattavasti piirikorttien paikalleen asentamiseen. Testaus ja vian paikannus tällaisessa toiminnallisessa testauksessa on hidasta ja vaatii järjestelmän ohjelmiston toiminnan tuntemusta.

Testausjärjestelmä kehitettiin nopeuttamaan ja helpottamaan piirikorttien testausta sekä vian paikantamista. Järjestelmä testaa automaatiolaitteen piirikortit, paikantaa mahdolliset viat ja tallentaa testitulokset tekstitiedostoon. Testausjärjestelmän vaatimuksina oli ensisijaisesti testauksen helppous, nopeus sekä käyttäjäystävällisyys. Järjestelmän prototyyppi suunniteltiin tilaajan käyttöön, mutta tulevaisuudessa se tullaan luovuttamaan myös piirikorttien valmistajan käyttöön, jolloin kaikki valmistajalta tulevat piirikortit on valmiiksi testattu.

Valmiilla järjestelmällä piirikortin testaus aloitetaan kytkemällä piirikortti virtalähteeseen. Piirikortilla oleva vihreä LED-valo syttyy, mikäli piirikortin virransyöttö toimii. Seuraavassa vaiheessa ladataan testiohjelma piirikortilla olevan mikrokontrollerin Flash-muistiin sarjaliikenneväylää pitkin. Jos testiohjelman lataaminen onnistuu, on sarjaliikenneväylä testattu toimivaksi. Testiohjelman lataaminen on mahdollista, jos piirikortin 60:sta signaalista 4 on toimivia: käyttöjännite, maa, sarjaliikenteen vastaanotto ja sarjaliikenteen lähetys. Tämän jälkeen testiohjelma testaa loput 60:sta signaalista.

2 TESTAUSJÄRJESTELMÄN SUUNNITTELUSSA KÄYTETYT OHJELMISTOT, MENETELMÄT JA TEORIA

Luvussa 2 esitellään opinnäytetyöprojektissa käytetyt ohjelmointikielet, ohjelmistot, teknologiat sekä niiden taustalla olevaa teoriaa.

2.1 Ohjelmiston organisointiparadigmat

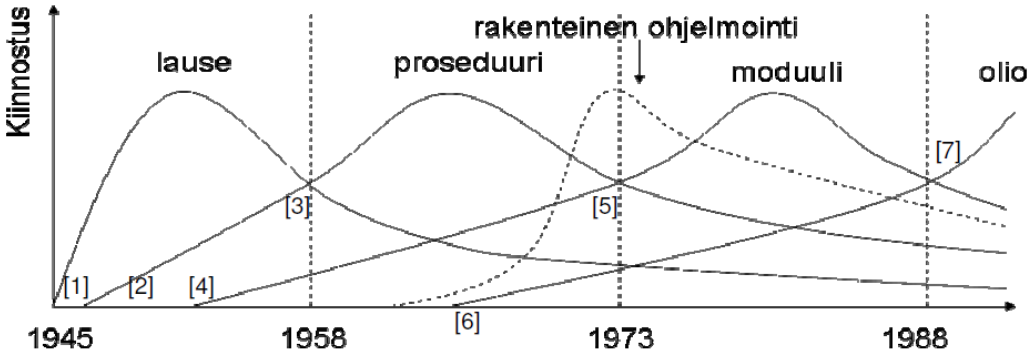
Ohjelmiston arkkitehtuurin toteuttamiseen on kehitetty erilaisia ajatusmalleja eli organisointiparadigmoja. Ohjelman kehityksessä noudatettavan organisointiparadigman valintaan vaikuttavat ohjelmiston laajuus, käyttötarkoitus ja ohjelmointikieli.

Ohjelmiston organisointiparadigmat voidaan jakaa lauseisiin, proseduureihin, modulaarisuuteen, olioihin sekä graafiseen ohjelmointiin perustuviin paradigmoihin. Lauseisiin ja proseduureihin perustuvat tietokoneohjelmat noudattavat rakenteetonta arkkitehtuuria. Niissä ohjelma sisältää yhden suoritettavan ohjelman, joka sisältää kaiken toiminnallisuuden. Rakenteeton tietokoneohjelma on hyvin alkeellinen lähestymistapa suunniteltaessa ohjelmistoja, eikä pelkkiin lauseisiin ja proseduureihin perustuvia ohjelmia ole juuri 1970-luvun alkupuolen jälkeen laajasti suunniteltu. (Tapaninen 2007, 5.)

Rakenteellisessa tietokoneohjelmassa ohjelmiston toiminnallisuus on jaettu useaan erilliseen toiminnalliseen yksikköön, moduuliin tai olioon. Tällöin laajan ohjelmiston tuotanto, virheenetsintä ja ylläpito helpottuvat ja ohjelmiston lähdekoodi säilyy ihmiselle havainnollisena. (Silander 1999.)

Luvussa 2.1 keskitytään esittelemään modulaarinen ja oliopohjainen organisointiparadigma, koska opinnäytetyössä kehitettyjen ohjelmien arkkitehtuurit noudattavat näitä lähestymistapoja. Kuvassa 1 on esitetty ohjelmiston orga-

nisointiparadigmojen kehitys ohjelmistokehityksen historian alkumetreiltä 1980-luvun loppupuolelle.



KUVA 1. Organisointiparadigmojen historia ohjelmistokehityksessä

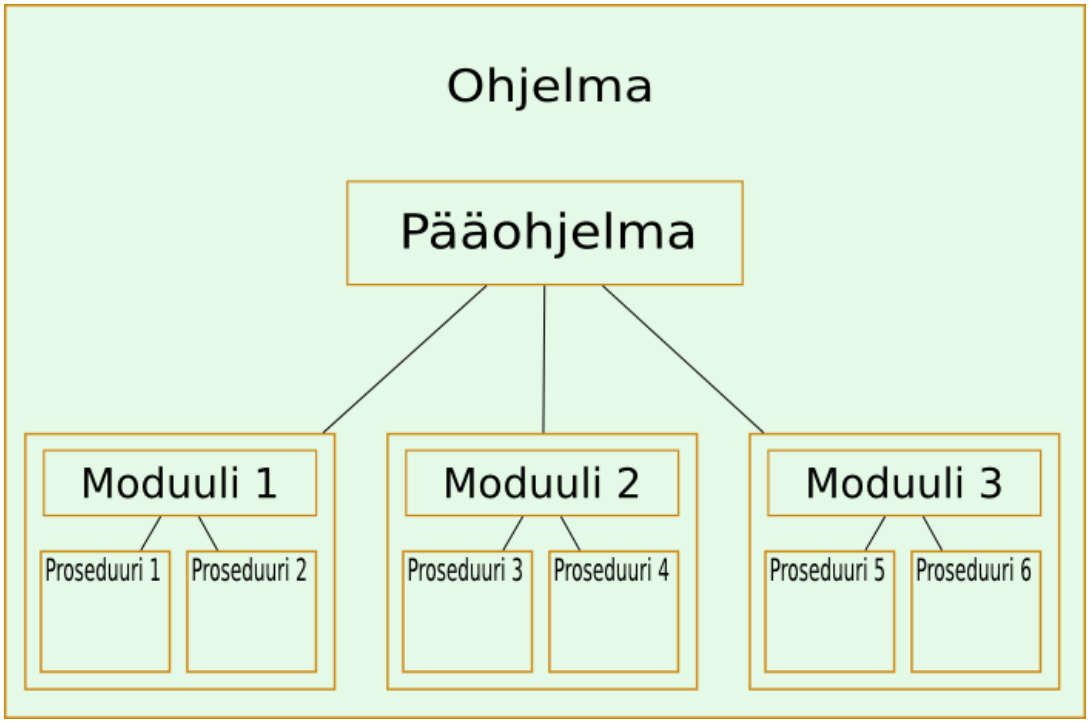
Kuvassa 1 ei ole esitetty graafisen ohjelmoinnin kehitystä, mutta se on ollut vahvasti mukana muun muassa testausympäristöjen ohjelmoinnissa 1980-luvun loppupuolen jälkeen. Graafisella ohjelmoinnilla tarkoitetaan ”drag and drop”
-tyyppistä ohjelmointia, jossa voidaan luoda esimerkiksi vuokaavio suoritettavasta ohjelmasta ja suunnitteluohjelma luo piirretystä vuokaaviosta lähdekoodin. Kaavioon liitettävät komponentit on toimivaksi testattuja ohjelmamoduuleja, joten ohjelmistosuunnittelija voi keskittyä ohjelmiston korkean tason ratkaisuihin. Graafinen ohjelmointi on nykyään laajasti käytössä muun muassa käyttöliittymien ohjelmoinnissa. (Tapaninen 2007, 5.)

2.1.1 Modulaarinen ohjelmointi

Testattavien piirikorttien mikrokontrollereille ladattava testiohjelma kirjoitettiin C-ohjelmointikielellä. Koska C-kieli ei tue oliopohjaista ohjelmointia, ohjelmiston arkkitehtuuri on suunniteltu siten, että se noudattaa modulaarisen tietokoneohjelman arkkitehtuuria. Tällöin ohjelman lähdekoodi on ihmiselle visuaalisesti havainnollinen, vaikka kyseessä olisi huomattavan laaja sovellus.

Modulaarisella ohjelmoinnilla tarkoitetaan ohjelmiston hajauttamista toiminnallisiin osiin eli moduuleihin. Modulaarinen ohjelmointi on yksi tapa suunni-

tella rakenteellinen ohjelmisto. Moduulit voivat sisältää suuren määrän pieniä proseduureja, joissa tietoa varsinaisesti muokataan. Proseduurit muokkaavat pääasiassa moduulin sisältämää paikallista tietoa. Kuvassa 2 on esitetty modulaarisen tietokoneohjelman rakenne. (Silander 1999; Tapaninen 2007, 10.)



KUVA 2. Modulaarisen ohjelman rakenne

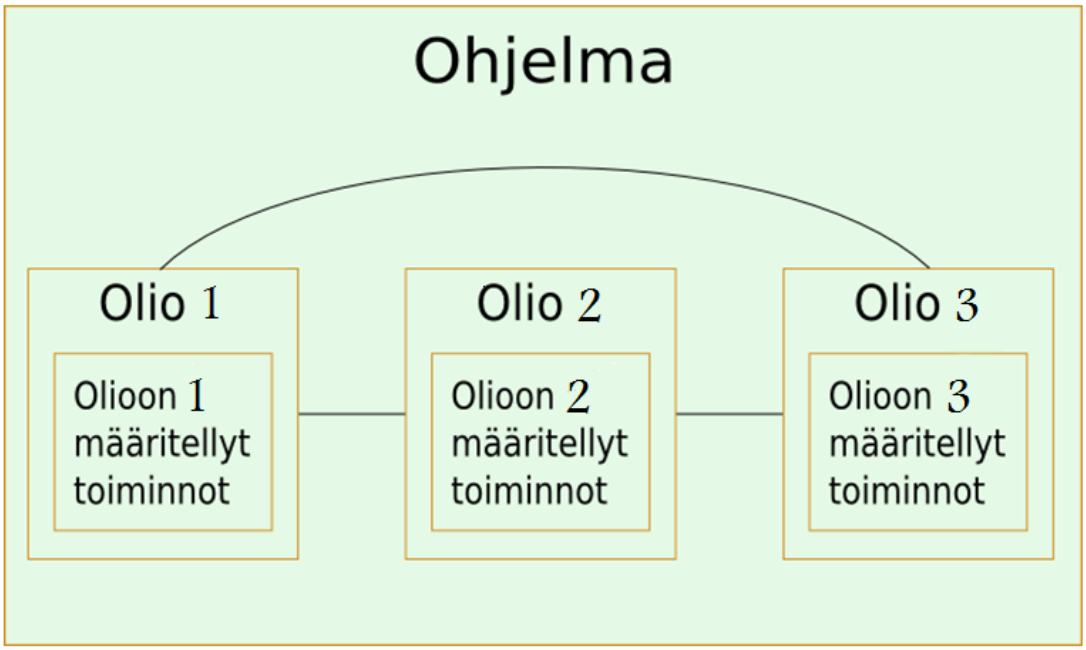
Modulaarisessa ohjelmassa ohjelmiston moduulit kommunikoivat pääohjelman kanssa rajapintojen kautta. Modulaarisen ohjelmiston ja oliopohjaisen ohjelmiston ero on, että oliopohjaisen ohjelman olio pystyy muokkaamaan myös toisen olion paikallista tietoa, toisin kuin moduuli, joka voi käsitellä vain omia paikallisia muuttujia sekä globaaleja muuttujia. Modulaarisessa C-kielisessä ohjelmassa moduulit ovat funktioita, jolloin moduulin rajapinta pääohjelmaan on funktion parametrit ja paluuarvot. (Silander 1999; Tapaninen 2007, 10.)

Toimivaksi testattuja moduuleja voidaan käyttää uudelleen muissa ohjelmissa, jolloin vältytään saman koodin uudelleen kirjoittamiselta. Myös virheenetsintä helpottuu, kun moduuleja voidaan testata erillään muista. Tällöin vika voidaan paikantaa tiettyyn moduuliin. (Silander 1999; Tapaninen 2007, 10.)

2.1.2 Olio-ohjelmointi

Testausjärjestelmän käyttöliittymäsovellus kirjoitettiin oliopohjaista ohjelmistoarkkitehtuuria tukevalla Python-ohjelmointikielellä. Olio-ohjelmointi on modulaarista menetelmää kehittyneempi tapa toteuttaa rakenteellinen, helposti muokattava ja ylläpidettävä sovellus.

Oliopohjaisessa ohjelmistossa olio on ohjelmiston toiminnallinen osa, joka voi sisältää muun muassa yksityisiä, yleisiä ja suojattuja muuttujia, funktioita ja proseduureja sekä aliolioita. Oliopohjainen tietokoneohjelma koostuu useiden olioiden vuorovaikutteisesta kokonaisuudesta, jossa jokaisella oliolla on oma rooli. Ohjelma on pilkottu useisiin itsenäisesti toimiviin osiin, jolloin kokonaisuuden hahmottaminen helpottuu. Oliot ovat merkityksellisiä myös kontekstista erotettuina, jolloin niitä voidaan käyttää hyväksi myös muissa sovelluksissa. Ohjelmoijan tulee tietää ainoastaan mitä olio tekee; teknisiin ratkaisuihin ei tarvitse ottaa kantaa. Kuvassa 3 on esitetty oliopohjaista arkkitehtuuria noudattavan ohjelman rakenne. (Tapaninen 2007, 11.)



KUVA 3. Oliopohjaisen ohjelman rakenne

2.2 Ohjelmointikielet

Testausjärjestelmän ohjelmien kirjoittamiseen käytettiin kahta erityyppistä ohjelmointikieltä. Testattavien piirikorttien mikrokontrollereille ladattavan testiohjelman ohjelmointikielenä käytettiin vanhempaa, nykyään matalan tason ohjelmointikieleksi luokiteltua C-ohjelmointikieltä. Käyttöliittymäsovelluksen ohjelmointiin valittiin uudempi, korkean tason tulkattava Python-ohjelmointikieli.

2.2.1 C-kieli

C-ohjelmointikieli on alun perin kehitetty Bellin Laboratoriossa vuonna 1972 UNIX-käyttöjärjestelmän ohjelmointikieleksi. 1970-luvun loppupuolella C-kieli oli tullut tunnetuksi ympäri maailman voimakkaana, joustavana ja hallitsevana ohjelmointikielenä ohjelmoijien keskuudessa, ja se alkoikin korvata sen aikaisia tunnettuja ohjelmointikieliä, kuten muun muassa PL/I:n ja ALGOL:n. (A Brief History of the C Language 2010.)

C-kielen vakautettua asemansa ohjelmointikielten maailmassa useat eri organisaatiot julkaisivat omia versioitaan C-kielestä. Tämä aiheutti suuria ongelmia ohjelmistosuunnittelijoille. Ongelman ratkaisuksi ANSI (American National Standards Institute) määräsi vuonna 1983 komitean vahvistamaan standardin määritelmän C-kielestä. Komitea hyväksyi ensimmäisen version standardista ANSI C-kielestä vuonna 1989. Vuonna 1990 myös ISO (International Standards Organization) hyväksyi ANSI C-kielen standardikseen. (A Brief History of the C Language 2010.)

Vaikka C-kieli eroaa huomattavasti ominaisuuksiltaan moderneista korkean tason ohjelmointikielistä, se on onnistunut säilyttämään asemansa yhtenä tunnetuimmista ja käytetyimmistä ohjelmointikielistä, erityisesti sulautettujen järjestelmien ohjelmoinnissa johtuen sen laiteläheisyydestä. Käytännössä kaikille tämän päivän 8- ja 16-bittisille mikroprosessoreille ja -kontrollereille on olemassa C-kääntäjä. (A Brief History of the C Language 2010.)

Koska C-kieli sisältää myös korkean tason ohjelmointikielen piirteitä, kuten mahdollisuuden kirjoittaa modulaarista organisointiparadigmaa noudattava sovellus, joten se sopii myös monimutkaisten sovellusohjelmien ohjelmointiin. C-kielinen ohjelma on oikein kirjoitettuna erittäin vakaa, nopea ja tehokas johtuen useista käytettävissä olevista tietotyypeistä ja vahvoista operaattoreista. (A Brief History of the C Language 2010.)

2.2.2 Python

Python-ohjelmointikielen kehityksen aloitti Guido von Rossum vuonna 1990. Siitä lähtien Pythonia on kehittänyt lukuisa joukko vapaaehtoisia ohjelmistokehittäjiä ja harrastelijoita. Pythonin kehitystyö noudattaa vapaan lähdekoodin periaatteita, ja se on vapaasti kaikkien ladattavissa, käytettävissä ja muokattavissa. (A Brief History of Python 2010.)

Python-ohjelmointikieli on tulkattava, interaktiivinen ja oliopohjaista ohjelmointia tukeva korkean tason ohjelmointikieli. Tulkattavuudella tarkoitetaan lähdekoodin ajon aikaista kääntämistä suoritettavaan muotoon. Lähdekoodia ei käännetä ennen ohjelman suorittamista, vaan Python-tulkki muodostaa lähdekoodista konekielistä ohjelmaa suorituksen aikana. (A Brief History of Python 2010.)

Python on yleiskäyttöinen korkean tason ohjelmointikieli, jota voidaan käyttää kaikissa moderneissa käyttöjärjestelmissä. Tärkein tekijä Pythonin valitsemiseksi testausjärjestelmän käyttöliittymän ohjelmointikieleksi oli mahdollisuus käyttää samaa lähdekoodia sekä Linux- että Windows-käyttöjärjestelmissä lähes muutoksitta. Ainoat muutokset mitä lähdekoodiin tuli tehdä sovitettaessa käyttöliittymäsovellusta Windows-ympäristöön, oli Linux-ympäristössä käytetyn sarjaportin polun muuttaminen Windowsin käyttämän com-portin numeroon. (A Brief History of Python 2010.)

2.2.3 wxPython

Testausjärjestelmän käyttöliittymäsovelluksen graafisten komponenttien ohjelmointiin käytettiin wxPython-grafiikkakirjastoa. WxPython on niin sanottu ”cross-platform toolkit” eli useassa eri käyttöjärjestelmässä toimiva kirjasto. Opinnäytetyön tekohetkellä wxPython-grafiikkakirjastoa tukevia käyttöjärjestelmiä ovat kaikki 32-bittiset Windows-käyttöjärjestelmät, lähes kaikki Unix- ja Linux-käyttöjärjestelmät sekä Macintosh OS X -käyttöjärjestelmät. (What is wxPython? 2010)

WxPythonin kehitystyö noudattaa Pythonin tapaan avoimen lähdekoodin periaatteita. Tukisivustoilta löytyy runsaasti koodiesimerkkejä graafisten komponenttien käyttöön. Nämä seikat vaikuttivat ratkaisevasti wxPythonin valintaan käytettäväksi grafiikkakirjastoksi. (What is wxPython? 2010)

2.3 Ohjelmistonkehitysympäristöt

2.3.1 Käyttöliittymäsovelluksen kehitysympäristö

Testausjärjestelmän käyttöliittymäsovellus kehitettiin alun perin toimimaan Linux-käyttöjärjestelmässä, joten ohjelmistonkehitys tapahtui pääasiassa Linux-ympäristössä. Windows-käyttöjärjestelmälle sovitettaessa ohjelmistoon tarvitsi tehdä vain hyvin pieniä muutoksia.

Ohjelmistonkehityksen työkaluina toimivat pääasiassa lähes kaikkien Linux-jakelupakettien mukana tuleva Nano-tekstieditori, Python-tulkki, wxPython-toolkit, dh-make-debian-paketointiohjelma sekä Windows-ympärisössä Py2Exe-tiedostonmuokkausohjelma. Virheenetsintään käytettäviä debuggaus-ohjelmia ei käytetty käyttöliittymän ohjelmistokehitystyössä, vaan ohjelmisto suunniteltiin siten, että ohjelmiston toiminnot pystyttiin testaamaan toimiviksi itsenäisinä osina ja lopulta liittämään yhteen toimivaksi kokonaisuudeksi. Valmis käyttöliittymäsovellus pakattiin Linux-käyttöjärjestelmille debian-pakettiin ja muokattiin Windows-käyttöjärjestelmille exe-tiedostoksi.

2.3.2 Testiohjelman kehitysympäristö

Mikrokontrollereille ladattavan testiohjelman kehitystyökaluina käytettiin mikrokontrollerien valmistajan suunnittelemaa integroitua ohjelmistonkehitysympäristöä, In-Circuit-Debuggeria sekä flash-muistin ohjelmointiin suunniteltua ohjelmaa.

2.4 Version hallinta

Ohjelmistonkehityksen tukena käytettiin versionhallintaohjelmia sekä käyttöliittymäsovelluksessa että testiohjelmassa. Versionhallinnalla tarkoitetaan menetelmää pitää kirjaa lähdekoodeihin tehdyistä muutoksista sekä tallentaa aikaisemmat kehitysversiot. Mikäli samaa ohjelmistoa kehittää useampi ohjelmistonkehittäjä, versionhallintaa käytettäessä vältytään tilanteelta, jossa kaksi tai useampi ohjelmistonkehittäjä tekee päällekkäisiä muutoksia lähdekoodeihin. Versionhallinnan lokitiedostoista pystytään katsomaan milloin, mihin ja kenen toimesta lähdekoodeihin viimeksi on tehty muutoksia.

2.4.1 Mercurial

Mercurial on GNU:n GPL-lisenssin alla julkaistu avoimen lähdekoodin tekstipohjainen versionhallintaohjelma. Se on kirjoitettu Python- ja C-ohjelmointikielillä alun perin Linux-käyttöjärjestelmille, mutta on myöhemmin portattu Windows- ja Mac OS-käyttöjärjestelmille. Mercurial eroaa muista versionhallintaohjelmista siten, että siinä ei ole yhtä varsinaista tietovarastoa (eng. repository), johon muutokset tallennetaan, vaan kehittäjät kloonaavat koko projektin kehityshistorian itselleen paikalliseksi kopioksi palvelimelta, ja tehdyt muutokset siirretään takaisin palvelimelle. Mercurialissa on myös sisäänrakennettu kevyt web-palvelin, jonka avulla tietovarastot voidaan jakaa nopeasti muiden kehittäjien kesken verkon yli. Mercurialia käytettiin opinnäytetyöprojektissa testausjärjestelmän käyttöliittymäsovelluksen ohjelmoinnin tukena Linux-ympäristössä. (http://linux.fi/wiki/Mercurial.)

2.4.2 Tortoise HG

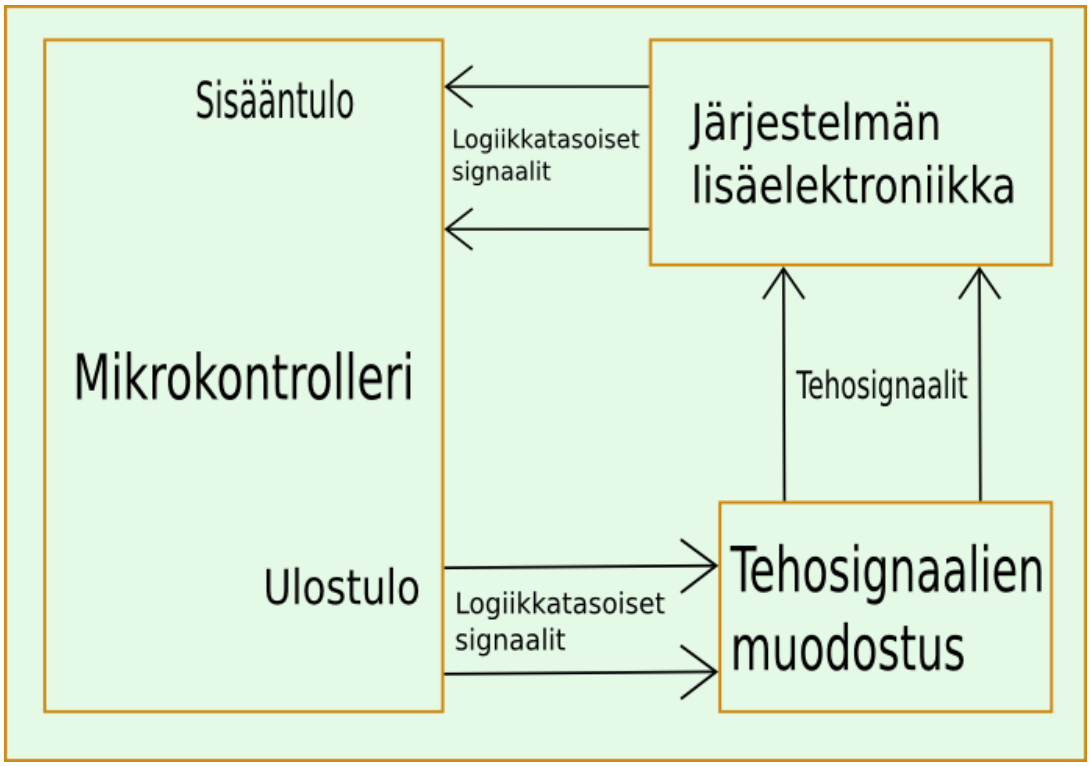
Tortoise HG on Mercurialin tietovarastoja käyttävä Windows-pohjainen graafisella käyttöliittymällä varustettu versionhallintaohjelma. Se on Mercurialin tapaan vapaasti kaikkien ladattavissa. Vaikka Mercurialia voidaan käyttää myös Windows-ympäristöissä, Tortoise HG otettiin käyttöön ohjelman graafisen käyttöliittymän tarjoamien hyötyjen selvittämiseksi. Sitä käytettiin opinnäytetyöprojektissa piirikorteille ladattavan testiohjelman kehityksen tukena Windows-ympäristössä. Tortoise HG sisältää käytännössä kaikki samat toiminnallisuudet kuin Mercurial mukaan lukien web-serverin. Graafisen käyttöliittymän tarjoamat hyödyt on pitkälti tottumiskysymys ja käyttäjän itse arvioitavissa. Opinnäytetyötä tehtäessä ei havaittu merkittäviä etuja verrattaessa graafista käyttöliittymää käyttävää Tortoise HG:ta tekstipohjaiseen Mercurialiin.

2.5 Testausjärjestelmän elektroniikka

Testausjärjestelmä pyrittiin suunnittelemaan siten, että testattavan piirikortin lisäksi tarvittaisiin mahdollisimman vähän ylimääräistä elektroniikkaa. Piirikorttien vaatimusmäärittelyn mukaisen testauksen mahdollistamiseksi testausjärjestelmään jouduttiin kuitenkin suunnittelemaan yksinkertainen lisäelektroniikkaa sisältävä piirikortti muun muassa tehosignaalien muuttamiseksi logiikkatasoisiksi signaaleiksi. Tässä kappaleessa esitellään elektroniikkasuunnittelun periaatteet sekä käytetyt työkalut.

2.5.1 Elektroniikkasuunnittelun periaatteet

Testausjärjestelmän lisäelektroniikan funktio on pääasiassa muuttaa kaksi eri taajuista tehosignaalia logiikkatasoisiksi siten, että mikrokontrolleri pystyy tulkitsemaan toimiiko tehosignaalien muodostus oikein. Kuvassa 4 on esitetty lisäelektroniikan sijoittuminen testausjärjestelmään.



KUVA 4. Testausjärjestelmän lisäelektroniikan sijoittuminen järjestelmään

Piirikortin testaus toimii pääsääntöisesti siten, että mikrokontrollerin ulostuloiksi ohjelmoidut I/O-liitännät on kytketty sisääntuloiksi ohjelmoituihin I/O-liitäntöihin. Tehosignaaleja ei kuitenkaan voida kytkeä suoraan mikrokontrollerin sisääntuloihin, vaan signaalit on muutettava ensin mikrokontrollerille sopivaksi logiikkatasoiseksi signaaliksi. Käytännössä tämä tarkoittaa tehosignaalien amplitudin maksimiarvon laskemista mikrokontrollerin käyttöjännitteen suuruiseksi.

Koska testattavan piirikortin elektroniikka tutkii valmiiksi, onko tehosignaalin muodostuksen ohjaussignaali halutulla taajuudella, lisäelektroniikan tehtäväksi jää tarkastaa, onko tehosignaali muodostunut. Elektroniikka on suunniteltu siten, että mikäli tehosignaali on muodostunut, mikrokontrollerille menevä logiikkatasoinen signaali on loogisessa 0-tilassa. Jos signaalia ei ole, eli tehosignaalin muodostuksessa on havaittu häiriö, mikrokontrollerille menevä logiikkatasoinen signaali on loogisessa 1-tilassa.

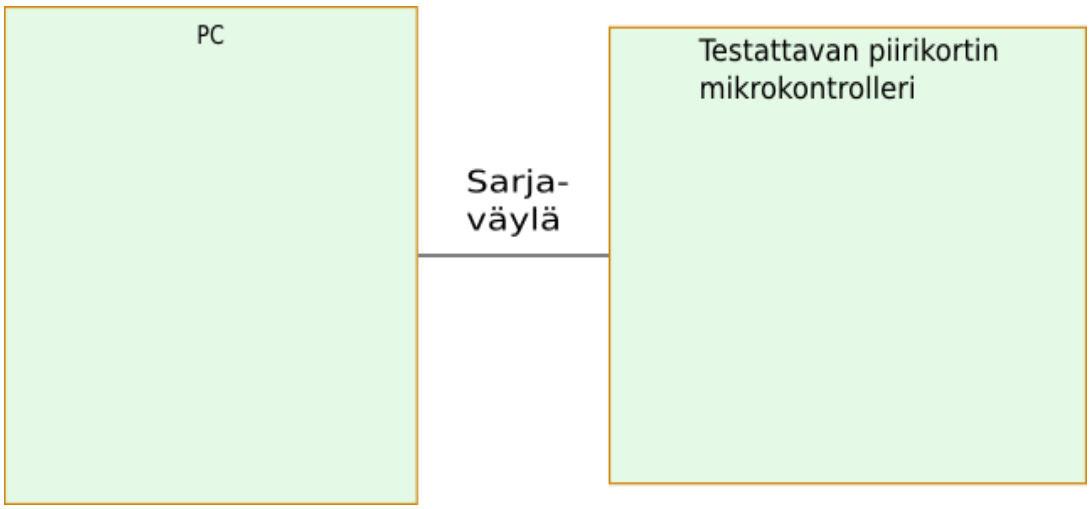
2.5.2 Suunnittelu- ja simulointityökalut

Testausjärjestelmän lisäelektroniikan suunnitteluun käytettiin Labcenter Electronicsin Proteus -piirikaavio- ja piirilevysuunnittelutyökaluja sekä ProSpice-simulointiohjelmaa. Prototyypin testaukseen käytettiin signaaligeneraattoria, tasavirtalähdettä sekä oskilloskooppia.

Piirikaavio suunniteltiin käyttäen komponenttien laskennallisia likiarvoja. Kytkennän toiminta varmistettiin spice-simuloinnilla. Simuloinnin jälkeen rakennettiin prototyyppilevy aksiaalikomponenteista. Prototyyppilevyä testattiin syöttämällä signaaligeneraattorilla testattavaa tehosignaalia vastaavaa signaalia kytkentään ja mittaamalla logiikkatasoisia ulostulosignaaleja oskilloskoopilla.

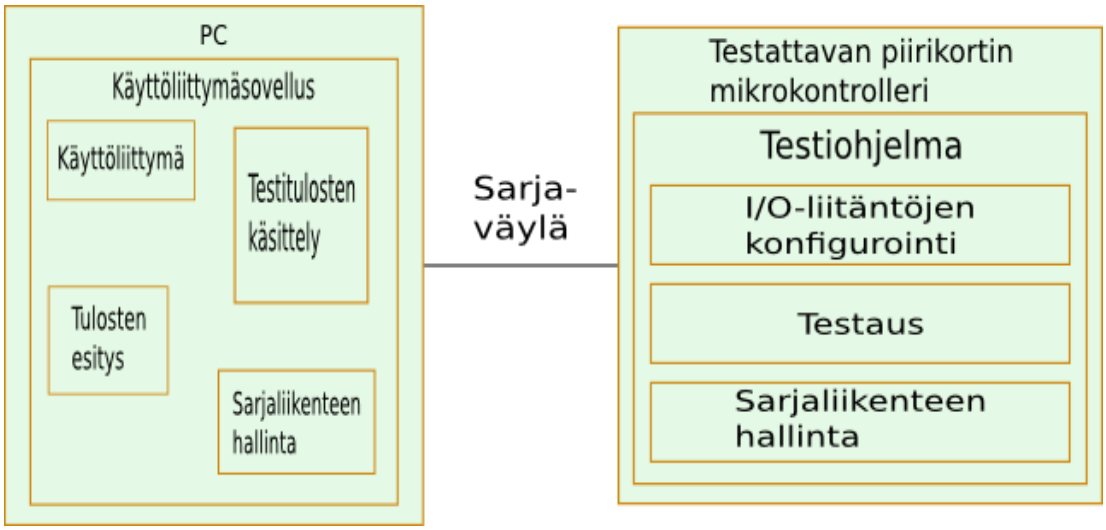
3 TESTAUSJÄRJESTELMÄN OHJELMISTON SUUNNITTELU

Testausjärjestelmän ohjelmiston suunnittelu aloitettiin hahmottelemalla lopullisen järjestelmän korkeimman tason lohkokaavio (kuva 5).



KUVA 5. Testausjärjestelmän korkeimman tason lohkokaavio

Järjestelmän korkeimman tason lohkoihin lisättiin ohjelmat sekä niiden pääasialliset toiminnalliset osiot. PC:n käyttöliittymäsovelluksen toiminnalliset osat toteutettiin Python-kielisen ohjelman olioina. Mikrokontrollerin ohjelmassa toiminnalliset osat toteutettiin siten, että yksi osio sisältää useita moduuleita eli funktioita, ja ne sijoitettiin omiin lähdekoodeihin. Kuvassa 6 on esitetty järjestelmän ohjelmiston toiminnalliset osat.



KUVA 6. Järjestelmän ohjelmiston toiminnalliset osat

Kuvassa 6 esitettyjen toimintojen lisäksi mikrokontrollerin ohjelma sisältää lisäksi useita pienempiä toimintoja, kuten testattavan piirikortin tyypin tarkastuksen ja testitulosten tallentamisen.

Ohjelmiston kehitys aloitettiin mikrokontrollereille ladattavan testiohjelman kirjoittamisesta. Koska mikrokontrollerien ohjelman rajapinta käyttöliittymäsovellukseen on sarjaliikenneväylä, ohjelma voitiin kirjoittaa kokonaisuudessaan valmiiksi ja testata sarjaterminaaliemulaattorilla ennen käyttöliittymäsovelluksen kirjoittamista.

Käyttöliittymäsovelluksen ohjelmointi aloitettiin graafisen komponenttien ohjelmoinnista. Kun graafiset komponentit saatiin halutunlaisiksi, ohjelmoitiin käyttöliittymän painonappien ja alasvetovalikoiden signaaleihin liitettävät tapahtumafunktiot. Käyttöliittymäsovellukseen kirjoitettiin huomattava määrä debuggauskoodia, jotta sen toiminta voitiin varmistaa ennen testiohjelman ja käyttöliittymäsovelluksen yhteistestausta.

3.1 Mikrokontrollerien ohjelma

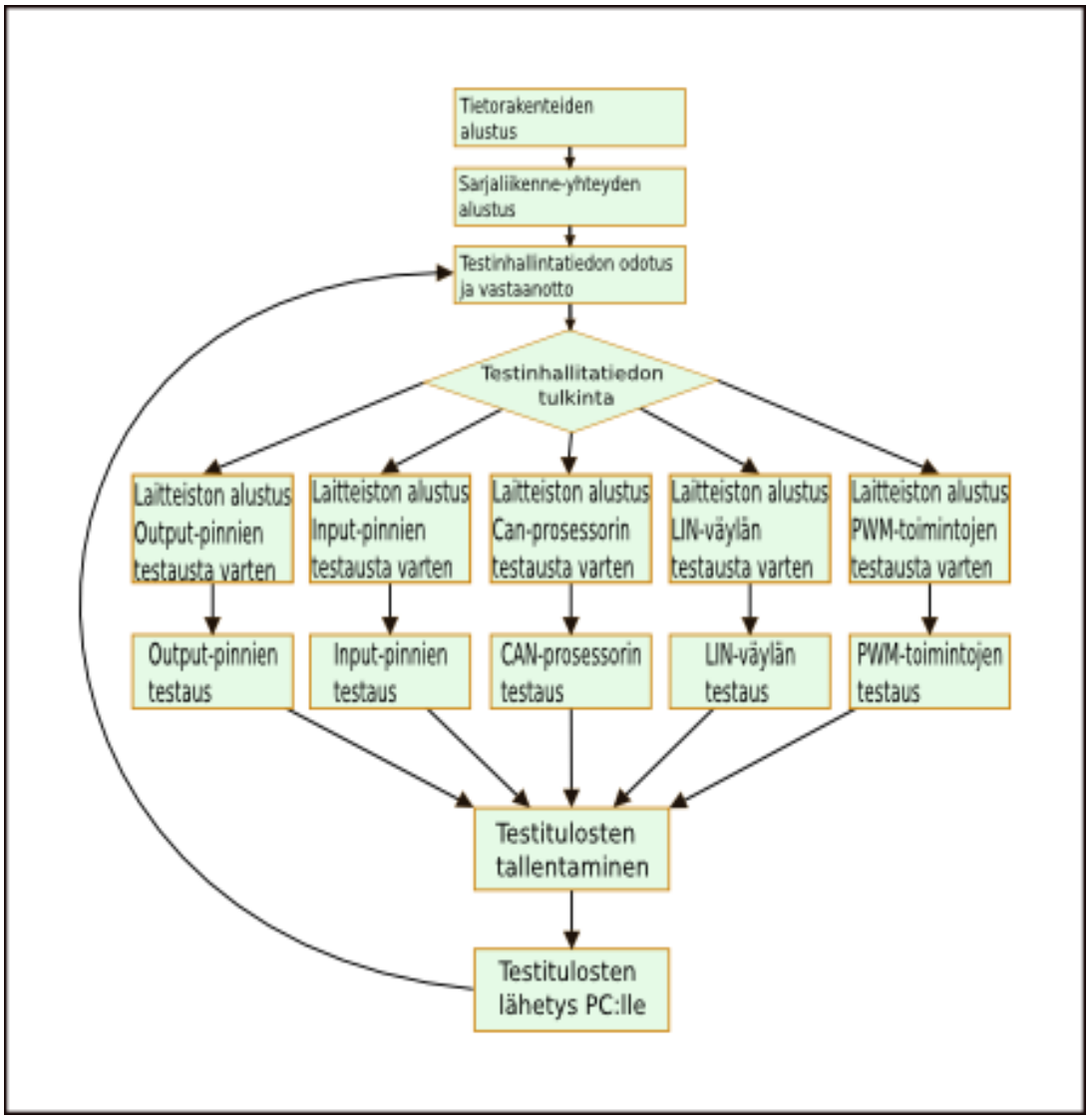
3.1.1 Suunnittelu ja toteutus

Piirikortilla käytetty mikrokontrolleri oli ennestään tuntematon, joten ohjelman suunnittelu aloitettiin tutustumalla mikrokontrollerin ominaisuuksiin kirjoittamalla pieniä moduuleita mikrokontrollerin valmistajan valmistamalle demokortille. Moduulit suunniteltiin siten, että samoja moduuleja voitiin käyttää hyväksi myöhemmässä vaiheessa varsinaisen testiohjelman suunnittelussa.

Piirikorttien testausjärjestelmän vaatimuksena oli, että järjestelmällä voidaan testata kahden erityyppisen piirikortin I/O-toiminnot. Testausjärjestelmän tuli olla samalla yksinkertainen ja helppokäyttöinen.

Koska molemmilla piirikorteilla on samanlainen mikrokontrolleri, ohjelma suunniteltiin siten, että sama image voidaan ladata molemman tyyppisille piirikorteille ja ohjelma testaa automaattisesti ennen testin aloittamista, kumman tyyppistä piirikorttia ollaan testaamassa. Tällä ratkaisulla vältytään tilanteelta, jossa testaaja lataa vahingossa väärän ohjelman testattavaan piirikorttiin ja näin olleen saa virheellisiä tuloksia testistä. Ratkaisu nopeuttaa myös huomattavasti piirikorttien ohjelmointia ennen testin aloittamista, sillä testaajan ei tarvitse vaihtaa ohjelmoitavaa imagea piirikorttisarjan ohjelmoimisen aikana.

Testiohjelman ajamisesta haluttiin tehdä nopeaa. Tämän takia toiminnot jaettiin siten, että mikrokontrollerien ohjelmassa suoritetaan vain välttämättömät toimenpiteet, ja kaikki muu toiminnallisuus on toteutettu PC:llä ajettavassa käyttöliittymäsovelluksessa. Ohjelman rungosta hahmoteltiin vuokaavio, joka on esitetty kuvassa 7.



KUVA 7. Mikrokontrollerin ohjelman vuokaavio

Testiohjelma pyörii ikuisessa silmukassa tutkien sarjaväylän RX-bufferin tilaa. Kun testinhallintakomento on vastaanotettu ja tulkattu, suoritetaan laitteiston konfigurointi yhden osion testausta varten ja testitulokset lähetetään sarjaväylään. Testiohjelma ei tutki testituloksia, vaan tulokset lähetetään suoraan PC:n käyttöliittymäsovellukseen, jossa testitulokset tulkitaan. Testitulosten lähetyksen jälkeen ohjelma palaa takaisin odottamaan testinhallintakomentoa eli tutkimaan RX-bufferin tilaa. Kun käyttöliittymäsovellus on tulkannut testitulokset, se lähettää uuden testinhallintakomennon testiohjelmalle. Testiohjelma suorittaa testinhallintakomentoa vastaavat toimenpiteet.

Testattaville osioille määriteltiin omat tietuemuuttujat, jotka sisältävät testattavan osion nimen ja testitulokset. Tietuemuuttujat ovat pääohjelman paikallisia muuttujia. Kaikille tietueille määriteltiin myös osoitinmuuttujat tietueiden viemiseksi parametrina funktioille. Globaalien muuttujien käyttöä pyrittiin välttämään tiedon suojaamisen helpottamiseksi. Osoitinmuuttujia käytettiin välttämään tietueiden kopiointi funktioon parametrina viemisen yhteydessä ja näin ollen nopeuttamaan ohjelmiston toimintaa. Kuvassa 8 on esitetty **tietorakenteiden alustus** lähdekoodissa.

```
struct test_data{
        char pin_name[8][8];
        char test_result[8][8];
        };

main()
        {
        struct test_data input_pins, *input_pins_ptr;
                        .
                        .
                        .
        struct test_data output_pins, *output_pins_ptr;
```

KUVA 8. Tietorakenteiden alustus

Ohjelman rajapinta PC:n käyttöliittymäsovellukseen on USB to RS-232 -sarjaväylä. **Sarjaliikenneyhteys** alustettiin baudinopeudelle 19200 bps/8,N,1, eli 8 databittiä, asynkroninen lähetys ja yksi stop-bitti. Sarjaliikenteen yhteydessä ei käytetty laitteistokeskeytyksiä, vaan RX-bufferin tilaa tutkitaan aina, kun edellisen testinhallintakomennon mukaiset toimenpiteet on suoritettu loppuun. Sarjaliikenteen lähetettävän ja saapuvan tiedon triggaamista keskeytyksistä ei olisi ollut hyötyä, sillä testiohjelma suorittaa joka tapauksessa edellisen komennon vaatimat toimenpiteet loppuun ennen uuden komennon tulkkaamista. Kuvassa 9 esitetyssä lähdekoodin osassa on esitetty RX-bufferin lippumuuttujan pollaus.

```
        .
        .
        .
while(!RXF);
byte_from_usart = RXBUF;
        .
        .
        .
```

KUVA 9. Sarjaliikenteen saapuvan tiedon pollaus

Ohjelma jää odottamaan while-silmukkaan niin pitkäksi aikaa, että RX-bufferiin saapuu testinhallintakomento ja RXF-lippumuuttuja saa arvon 1. Tämän jälkeen luetaan RXBUF-rekisterin arvo.

Käyttöliittymäsovellus lähettää testiohjelmalle **testinhallintakomentoja**, jonka mukaiset tehtävät testiohjelma suorittaa. Käyttäjän aloitettua testin, ensimmäinen komento on piirikortin tyypin testaus. Kuvassa 10 olevassa lähdekoodin osassa on esitetty piirikortin tyypin testaus.

```
switch(test_control_data)
        {
        .
        .
        .
        case TEST_BOARD:
        init_hw(TYPE_TEST);
        send_board_type(test_board_type());
        break;
        .
        .
        .
        }
```

KUVA 10. Piirikortin tyypin testaus

Switch-valintalauseessa tutkitaan käyttöliittymäsovellukselta vastaanotettuja testinhallintakomentoja. Jos komento on TEST_BOARD, kutsutaan laitteiston alustavaa init_hw()-funktiota parametrilla TYPE_TEST, eli piirikortin tyypin testaus. Test_board_type()-funktio antaa piirikortille tyypin testaukseen vaadittavat testiherätteet ja tutkii vasteet. Funktio palauttaa piirikortin tyypin, joka lähetetään käyttöliittymäsovellukselle send_board_type()-funktiossa. Käyttöliittymäsovelluksen tulee tietää testattavan piirikortin tyypin, jolloin se tietää mitkä testinhallintakomennot sen tulee lähettää piirikortille. Piirikortin tyypin testauksen jälkeen loput testinhallintakomennot ovat piirikortin eri osien testinaloituskäskyjä. Kuvassa 11 on esitetty testinhallintakäskyn tulkinta ja sitä seuraavat toimenpiteet input-pinnien testauksessa.

```
switch(test_control_data)
        {
        .
        .
        .
        case INPUT_PINS:
        init_hw(INPUT_PINS);
        test_stimulus(INPUT_PINS, input_pins_ptr);
        send_results(input_pins_ptr);
        break;
        .
        .
        .
        }
```

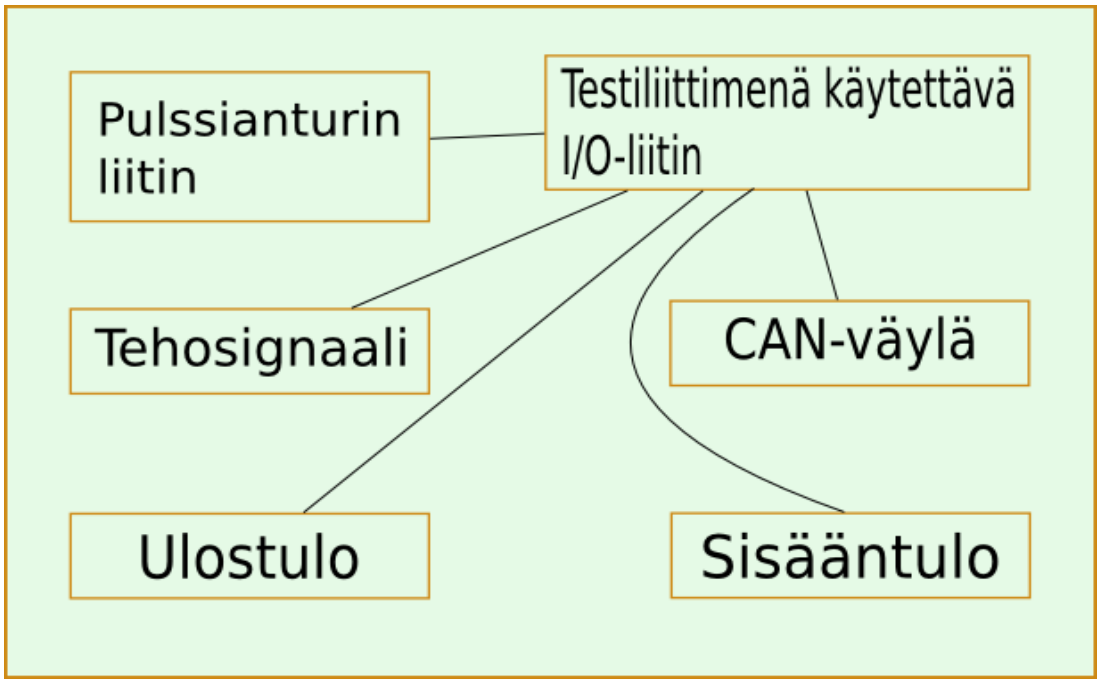
KUVA 11. Testinhallintatiedon tulkinta ja sitä seuraavat toimenpiteet

Testinaloituskäskyt tutkitaan samassa switch-valintalauseessa kuin piirikortin tyypin testaus. Kun käyttöliittymäsovellus lähettää piirikortille input-pinnien testauksen aloittavan testinhallintakomennon, testiohjelma alustaa laitteiston input-pinnien testausta varten. Test_stimulus()-funktiossa annetaan input-pinnien testauksen vaatimat testiherätteet, tutkitaan vasteet sekä tallennetaan tulokset.

Test_stimulus()-funktio ottaa parametreina testattavan osion sekä sitä vastaavan tietuemuuttujan osoittimen. Test_stimulus()-funktiosta kutsutaan edelleen save_results()-funktiota, joka tallentaa testitulokset tietueihin. Sa-

ve_results()-funktio ottaa parametrina tietuemuuttujan osoittimen, joka annettiin parametrina test_stimulus()-funktioon. Lopuksi testitulokset lähetetään käyttöliittymäsovellukselle funktiolla send_results(), joka ottaa jälleen parametrina testatun piirikortin osiota vastaavan tietuemuuttujan osoittimen. Send_results()-funktio lähettää tietueen sisältämän tiedon merkkijonoina sarjaporttiin.

Piirikorteille ei ole suunniteltu erillisiä osia testausta varten, vaan piirikortin eri osien testaus suoritetaan olemassa olevia I/O-liittimiä hyväksi käyttäen. Piirikortin osien testaus suunniteltiin siten, että yksi liitin valittiin testiliittimeksi, jonka **I/O-pinnejä konfiguroidaan** testattavan osion mukaan joko input- tai output-pinneiksi. Laitteisto alustetaan jokaisen osion testinaloituskäskyn jälkeen sen vaatimaan tilaan. Kuvassa 12 on esitetty testiliittimenä käytettävän liittimen kytkentä piirikortilta testattaviin osioihin.



KUVA 12. Testiliittimen kytkentä testattaviin osioihin

Piirikorteille suoritettava testi on tyypiltään Stuck-At-testi, eli testattavan liittimen pinneihin syötetään testiliittimen kautta loogiset 0- ja 1-tila ja tutkitaan, muuttuuko testattavan liittimen tila. Mikäli liittimen pinni on juuttunut 0-

tai 1-tilaan, kirjoitetaan testattavan osion tietueeseen kyseessä olevat pinnin kohtaan havaitun virheen tyyppi eli Stuck-At 0 tai Stuck-At 1.

LIN-väylän liitin testataan konfiguroimalla prosessorin LIN-väylää käyttävät pinnit normaaleiksi ulostulopinneiksi ja testiliittimen pinnit sisääntuloiksi. Tämän jälkeen testaus suoritetaan syöttämällä testiherätteet LIN-väylän pinneistä ja lukemalla vasteet testiliittimestä.

CAN-väylän normaali toiminnallinen testaus vaatisi toisen CAN-väylää käyttävän laitteen, joten CAN-väylän testaus tehdään konfiguroimalla CAN-prosessori loop-back-moodiin, jolloin väylälle lähetettävä tieto voidaan lukea suoraan saapuvan tiedon pinnistä. Tällä pystytään testaamaan mikrokontrollerin CAN-prosessorin toiminta. Mikäli CAN-väylän toiminnassa havaitaan läpi menneen testin jälkeen vika, se voidaan paikantaa signaalin differentiaaliseksi muuttavaan mikropiiriin.

Testausjärjestelmän elektroniikka on suunniteltu siten, että tehosignaalit voidaan testata lukemalla kahden testiliittimessä olevan sisääntuloksi konfiguroidun pinnin tilat. 0-tila vastaa toimivaa signaalia ja 1-tila häiriötä signaalissa.

3.1.2 Testaus

Testiohjelma suunniteltiin siten, että ohjelman osiot ja moduulit voitiin testata erillään kokonaisuudesta. Jokaisesta osiosta kirjoitettiin ensin debuggausversio, joka testattiin toimivaksi. Osioiden debuggausversiot sisältävät jonkin verran varsinaiseen testiohjelmaan kuulumatonta koodia.

Sarjaliikenteen hallinta oli ensimmäinen kirjoitettava testiohjelman osio, sillä sitä käytettiin kaikkien muiden osioiden testauksessa apuna. Sarjaliikenteen hallinta -osio sisältää sarjaliikenteen lähetykseen ja vastaanottoon vaadittavat moduulit, eli funktiot. Osion debuggausversio kirjoitettiin siten, että ohjelmaan syötettäessä sarjaterminaaliemulaattorilla ASCII-merkkejä ohjelma kaiuttaa saman merkin takaisin sarjaterminaaliin. Näin voitiin varmistaa sekä vastaanoton että lähetyksen toiminta. Lopullinen testiohjelmassa käytettävä

versio sarjaliikenteen-hallinta -osiosta sijoitettiin serial.c- ja serial.h-tiedostoihin.

Sarjaliikennettä hallinnoivan osion testauksen jälkeen kirjoitettiin osio kontrolloimaan **I/O-liitäntöjen konfigurointia**. Tämän osion kirjoittamiseen käytettiin hyväksi mikrokontrolleriin tutustumisen yhteydessä kirjoitettuja ohjelmamoduuleita.

Varsinaisessa testiohjelmassa I/O-liitännät konfiguroidaan testinhallintakomentojen perusteella joko input- tai output-liitännöiksi. Debuggausversiossa käytettiin sarjaliikenteen hallinta -osiota lähettämään debuggaustietoa sarjaterminaaliin.

Sarjaterminaalin kautta lähetetään testinhallintakomento I/O-liitäntöjen konfigurointia hallinnoivan osion deuggausversiolle. Testinhallintakomento kertoo osiolle, mikä osa piirikortista tullaan testaamaan. Debuggausversio tulkitsee vain piirikortin input-liittimen testauksen aloittamiseen varatun testinhallintakomennon. Osion debuggausversio konfiguroi testinhallintakomennon saatuaan piirikortin input-liittimen pinnit input-tyyppisiksi. Tämän jälkeen tutkitaan liittimen pinnejä vastaavan port data direction -rekisterin tilan ja tulkitaan, onko liittimen pinnit konfiguroitu oikeaan tilaan. Lopuksi sarjaterminaaliin lähetetään tieto siitä, onko pinnien konfigurointi onnistunut halutulla tavalla. Konfigurointi-osio sijoitettiin testiohjelmassa io_config.c- ja io_config.h-tiedostoihin.

Testiohjelmassa **testaus ja tulosten tallennus** -osio syöttää herätteet ja mittaa vasteet testattavasta liittimestä sekä tallentaa tuloksen tietueihin. Testaus ja tulosten tallennus -osion debuggausversio kirjoitettiin siten, että testauksen logiikka voitiin testata syöttämättä herätteitä todellisten liittimien kautta. Tämä toteutettiin käyttämällä todellisten I/O-porttien datarekistereiden sijaan ohjelmallisia muuttujia. Tämän osion debuggausversioon kirjoitettiin lopullisen testiohjelman pääfunktion tietuemuuttujat, joihin I/O-porttien datarekistereitä mallintavien muuttujien tilat tallennetaan vertailuoperaatioiden jälkeen. Sarjaliikenteen hallinta -osiota käytettiin lähettämään tietuemuuttujiin tallennettuja testituloksia sarjaterminaaliin.

Kun logiikka saatiin toimimaan ohjelmallisilla muuttujilla, testaus ja tulosten tallennus -osioon yhdistettiin I/O-liitäntöjen konfigurointi -osio ja testaus suoritettiin syöttäen testiherätteitä testiliittimenä käytetyn I/O-liittimen kautta testattavaan liittimeen ja lukemalla liittimen pinnien tilat. Tulokset tallennettiin tietuemuuttujiin ja lähetettiin sarjaterminaaliin.

Kun kaikki kolme toiminnallista osiota oli testattu debuggausversioilla, ne **liitettiin yhteen** toimivaksi kokonaisuudeksi. Tässä vaiheessa testiohjelma otti vastaan yhden testinhallintakomennon, jonka tulkittuaan suoritti input-liittimen testauksen. Tässä vaiheessa ohjelman runko oli valmis, ja kaikki ohjelman osiot oli testattu toimiviksi. Kehityksen seuraavissa vaiheissa ei käytetty enää osioiden debuggausversioita, vaan loput ohjelman ominaisuudet, eli piirikortin kaikkien liittimien testaukseen vaadittava koodi, kirjoitettiin yhteen liitettyjen osioiden jatkoksi.

Koska testiohjelma palaa suoritettuaan testinhallintakomentoa vastaavat tehtävät odottamaan seuraavaa testinhallintakomentoa, ohjelmistoon oli selkeää ja yksinkertaista lisätä loput ominaisuudet piirikortin kaikkien osien testaamista varten. Kehitys lopulliseen versioon eteni siten, että osioiden yhteenliittämisen jälkeen ohjelmaan lisättiin yhden piirikortin osan testaukseen vaadittava koodi ja se testattiin toimivaksi ennen seuraavan lisäämistä. Ohjelmisto suunniteltiin tällä tavalla, jotta tulevaisuudessa piirikorttien muuttuessa testiohjelmaan voidaan helposti lisätä ja poistaa piirikortin osien testaukseen vaadittavia osia.

3.2 Käyttöliittymäsovellus

Piirikorttien testausjärjestelmän käyttöliittymäsovellus kirjoitettiin Python-ohjelmointikielellä. Python oli opinnäytetyön tekijälle projektin alkuvaiheessa täysin tuntematon ohjelmointikieli, joten käyttöliittymän kehitys oli alussa kohtuullisen hidasta ja haastavaa. Koska Python on kuitenkin syntaksiltaan erittäin selkeä ja nopeasti opittava ohjelmointikieli, edistystä tapahtui heti alusta lähtien. Opinnäytetyön tekijän aikaisempi kokemus korkean tason ja käyttöliittymän ohjelmoinnista edesauttoi huomattavasti Python-ohjelmointikielen oppimista.

3.2.1 Suunnittelu ja toteutus

Testausjärjestelmän ohjelmiston vaativimmat toiminnot toteutettiin käyttöliittymäsovelluksessa. Sovelluksen suunnittelu ja toteutus aloitettiin opettelemalla käytettävän ohjelmointikielen syntaksi. Tämä tapahtui kirjoittamalla muutamia yksinkertaisia konsolisovelluksia, jotka käyttivät tavanomaisia valinta- ja toistorakenteita.

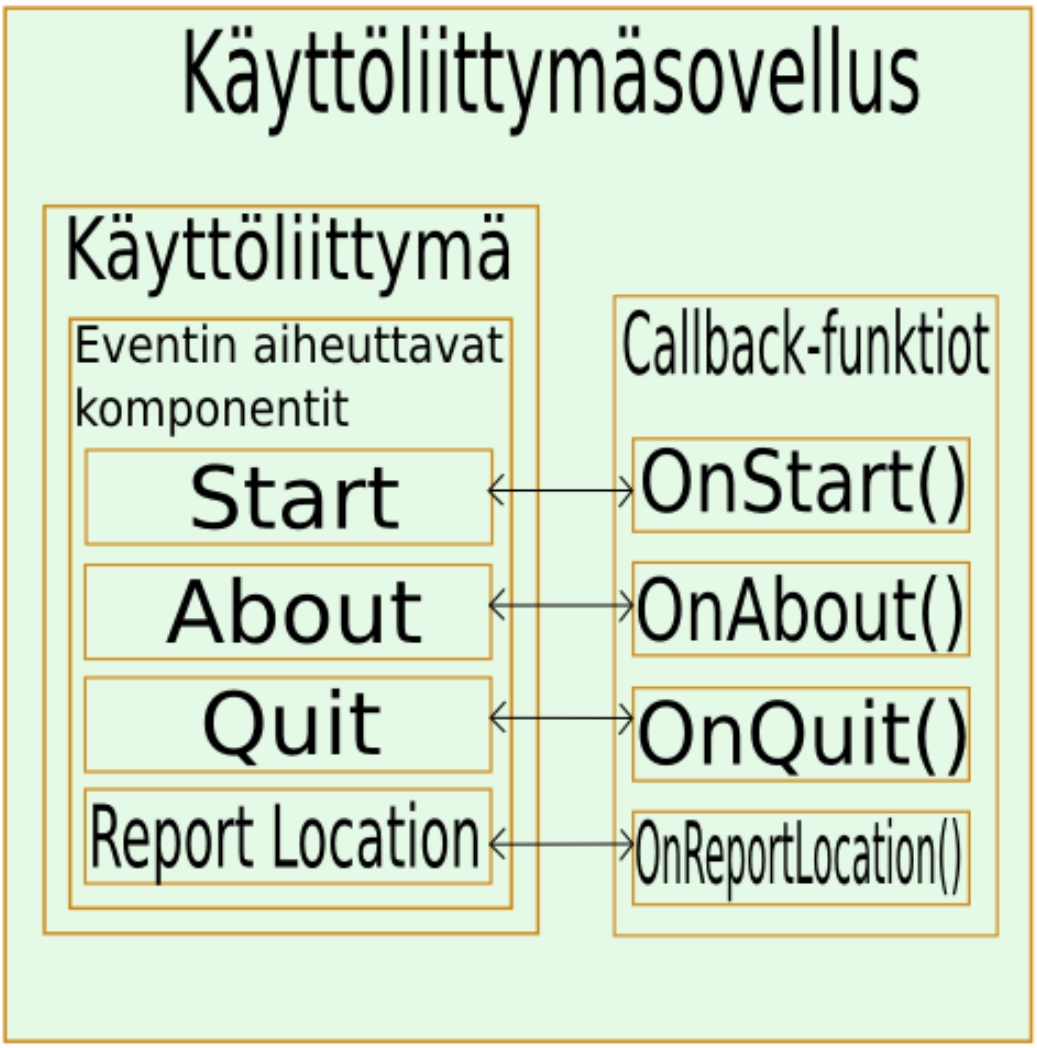
Kun Python-kielisen ohjelman lähdekoodin kirjoittaminen alkoi tuntua luontevalta, otettiin käyttöön wxPython-grafiikkakirjasto ja kirjoitettiin muutamia yksinkertaisia graafisia komponentteja sisältäviä harjoitusohjelmia. Kun graafisten osien ohjelmointi wxPythonilla alkoi sujua, aloitettiin varsinaisen testausjärjestelmän käyttöliittymäsovelluksen ohjelmointi.

Käyttöliittymä on käyttäjäystävällisyyteen eniten vaikuttava tekijä, joten se suunniteltiin niin yksinkertaiseksi ja selkeäksi kuin mahdollista. Ongelmana kuitenkin on, että mitä yksinkertaisempi kohtalaisen laajan järjestelmän käyttöliittymä on, sitä monimutkaisemmaksi ja laajemmaksi muodostuu muu ohjelmisto. Käyttöliittymän suunnittelu aloitettiin hahmottelemalla halutunlainen ulkoasu vektorigrafiikkaohjelmalla, joka on esitetty kuvassa 13.

KUVA 13. Hahmoteltu käyttöliittymä

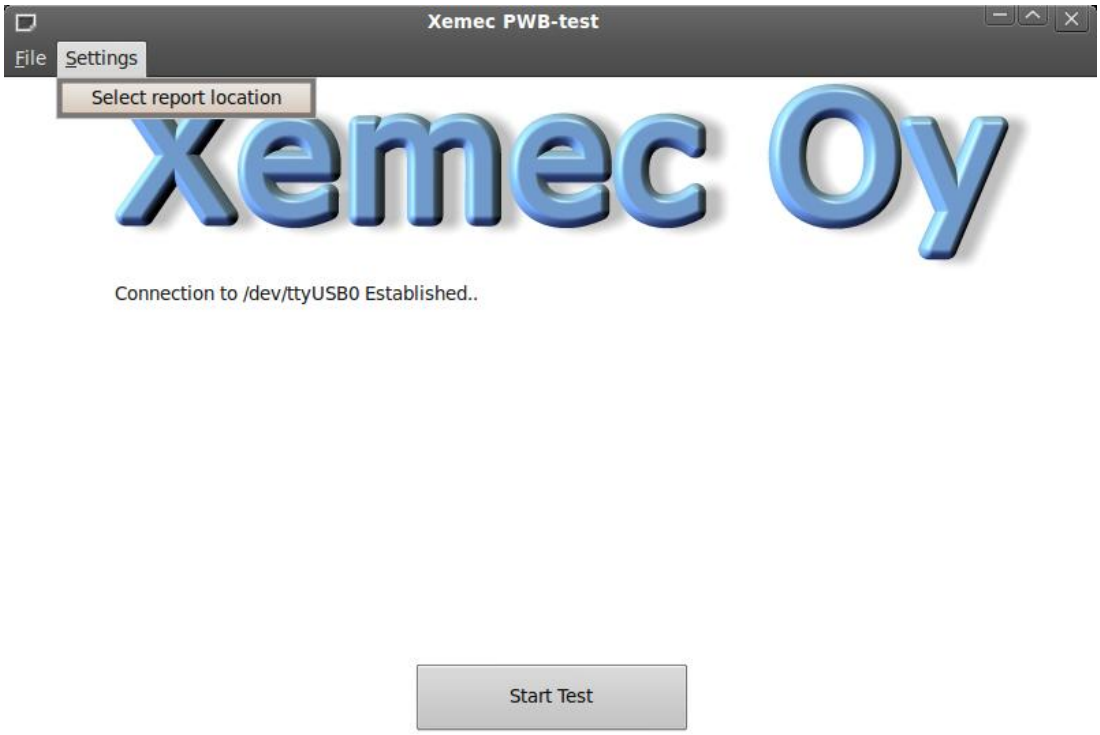
Kuvasta 13 nähdään, että käyttöliittymän haluttiin sisältävän painonappi testin aloittamista varten ja tekstikenttä testitulosten ja testatun piirikortin tyypin esittämistä varten. Käyttöliittymän yläpaneelissa on kaksi valikkoa, joista File-valikossa on ohjelman lopetusta varten Quit-valinta ja ohjelman tietoja varten About-valinta. Settings-valikossa on Report Location -valinta, jolla voidaan valita testitulosten tallennuspaikka, mikäli tulokset halutaan tallentaa tiedostoon.

Käyttöliittymä voitiin kehittää täysin valmiiksi erillään muusta toiminnallisuudesta, koska toiminnallisuudet on sidottu painonappien ja vetovalikoiden klikkaamisen aiheuttamiin tapahtumiin eli eventteihin. Käyttöliittymästä tehtiin ohjelmallinen olio. Kuvassa 14 on esitetty käyttöliittymän komponenttien suhde callback-funktioihin.



KUVA 14. Käyttöliittymän suhde callback-funktioihin

Käyttöliittymän ohjelmointi aloitettiin ohjelmoimalla käyttöliittymän pääikkuna, eli main window. Pääikkuna sisältää kaksi wxPanel-komponenttia. Ne ovat ikään kuin näkymättömiä ali-ikkunoita, jotka helpottavat näkyvien komponenttien sijoittelua haluttuihin paikkoihin. Ylempään wxPanel-komponenttiin ohjelmoitiin yrityksen logo ja alempaan tekstikenttä testitulosten esittämistä varten sekä painonappi testin aloittamista varten. Kuvassa 15 on esitetty testausjärjestelmän lopullinen käyttöliittymä.



KUVA 15. Lopullinen käyttöliittymä

Kun käyttöliittymän graafiset komponentit saatiin ohjelmoitua ja sijoitettua haluttuihin paikkoihin, käyttöliittymän vetovalikoiden ja painonapin aiheuttamiin eventteihin liitettiin niin sanotut callback-funktiot. Callback-funktiota kutsutaan aina, kun siihen liitetty graafinen komponentti aiheuttaa eventin.

Käyttöliittymän toiminta testattiin kirjoittamalla jokaiselle eventin aiheuttavalle graafiselle komponentille oma debuggaus-callback-funktio, joka tulosti tekstiä standardiin ulostuloon. Näin voitiin testata, että kaikki eventit aiheuttavat callback-funktion kutsun. Alla olevassa kuvassa on esitetty callback-funktion liittäminen komponentin aiheuttamaan eventtiin.

```
self.button.Bind(wx.EVT_BUTTON, self.OnStart)
```

KUVA 16. Callback-funktion liittäminen komponentin aiheuttamaan eventtiin

Self.button-niminen komponentti liitetään OnStart()-nimiseen call-back-funktioon Bind-metodilla. Parametrina oleva wx.EVT_BUTTON kertoo, minkä tyyppisen eventin self.button-komponentin klikkaaminen aiheuttaa.

```
def OnStart(self,event):
        print "Start-button clicked"
```

KUVA 17. Start-painonappiin liitettävä debuggaus-callback-funktio

Kuvassa 17 on esitetty start-painonappiin liitettävä debuggaus-callback-funktio. Mikäli käyttöliittymän start-painonappia painetaan, funktio tulostaa standardiin ulostuloon Start-button clicked -merkkijonon. Tällä voidaan varmistaa, että callback-funktion liittäminen eventtiin on onnistunut.

Valmiissa sovelluksessa testin käynnistävää painonappia painettaessa kutsutaan OnStart()-nimistä callback-funktiota, jossa suoritetaan kaikki varsinaisesti piirikortin testaukseen liittyvät toimenpiteet.

Report location -valintaa painettaessa kutsutaan OnReport-callback-funktiota, jossa avataan uusi ikkuna. Ikkunassa näkyvät PC:n kaikki kansiot ja tiedostot, joista voidaan valita tiedosto testitulosten tallentamista varten.

About-valintaa painettaessa kutsutaan OnAbout-callback-funktiota, jossa luodaan uusi ikkuna. Siinä näkyy ohjelman versio ja päivämäärä sekä kehittäjän yhteystiedot vikojen raportoimista varten. Ikkunassa on painonappi, jota painamalla ikkuna tuhoutuu.

Quit-valintaa painettaessa kutsutaan OnQuit()-callback-funktiota, jossa suljetaan sarjaliikenneyhteys, suljetaan mahdollinen tiedosto, johon testituloksia tallennetaan ja tuhotaan sovelluksen pääikkuna sekä suljetaan sovellus. Valmis käyttöliittymä sijoitettiin Gui.py-tiedostoon.

Käyttöliittymäsovelluksen **sarjaliikenteen hallinta** -osa kirjoitettiin puhtaasti Python-kielisen ohjelman oliona, jolloin varsinaista debuggausversiota ei tarvittu, vaan olion toiminta voitiin testata olioiden testaukseen kirjoitetussa debuggausohjelmassa. Siitä kerrotaan enemmän luvussa 3.2.2.

Sarjaliikenteen hallinta -olion ohjelmointi aloitettiin kirjoittamalla siihen funktio sarjaliikenneyhteyden avaamiseksi. Funktio ottaa parametrina avattavan sarjaportin polun Linux-versiossa ja sarjaportin numeron Windows-versiossa. Mikäli sarjaportin avaaminen ei onnistu, funktio antaa virheilmoituksen standardiin ulostuloon.

```
if(ser.isOpen == FALSE)
        print "Can't open serial port"
```

KUVA 18. Sarjaportin tilan tutkiminen

Sarjaporttien polut on annettu alustusarvoina Linux-versiossa porttiin /dev/ttyUSB0 ja Windows-versiossa porttiin COM2.

Sarjaliikenteen avaamisen jälkeen olioon lisättiin sarjaportin lukemisen ja kirjoittamisen mahdollistavat funktiot. Sarjaportin lukemista ja kirjoittamista testattiin piirikortilla, jolle oli ladattu testiohjelma. Sarjaliikenteen hallinta -oliolla lähetettiin testinhallintakäsky piirikortille ja luettiin kortin palauttamat testitulokset. Kun sarjaliikenteen lukeminen ja kirjoittaminen saatiin toimimaan, olioon lisättiin lopuksi funktio sarjaportin sulkemiseksi. Valmis sarjaliikenteen hallinta -olio sijoitettiin SerialControl.py-tiedostoon.

Testitulosten käsittely ja esitys -osasta kirjoitettiin oma olio. Olioon kirjoitettiin funktiot merkkijonon tutkimista ja testitulosten esitystä varten. Olion testaus vaati käyttöliittymäsovelluksen läsnäolon, sillä olion sisältämä toinen funktio tulostaa testituloksia käyttöliittymän tekstikenttään. Merkkijonojen tutkimiseen kirjoitettua funktiota testattiin syöttämällä testausalustan globaaleihin muuttujiin sijoitettuja merkkijonoja funktiolle ja tutkimalla funktion paluuarvoja.

Merkkijonon tutkiminen tapahtuu siten, että merkkijonosta tutkitaan ensimmäisenä, minkä osion testituloksia ollaan tutkimassa. Tämän jälkeen merkkijonoa verrataan toimivan piirikortin kyseessä olevan osion tuottamaan referenssijonoon. Mikäli jonot täsmäävät, funktio palauttaa ”Ok”-arvon. Jos merkkijonot eivät täsmää, eli piirikortin testattavassa osiossa on havaittu vika, merkkijonosta etsitään poikkeava kohta ja tutkitaan siinä oleva merkki. Merkki kertoo, onko kyseessä stuck-at 0- vai stuck-at 1 -vika. Merkkijonon indeksin perusteella tiedetään, missä fyysisessä pinnissä vika on havaittu. Mikäli vika on havaittu, funktio palauttaa merkkijonona testatun piirikortin osion, pinnien nimet, joissa vika on havaittu, sekä vian tyypin.

Testitulosten esitys -funktio tulostaa testitulokset käyttöliittymän tekstikenttään ja käyttäjän valitessa avaa ja tallentaa tulokset myös tekstitiedostoon. Tätä funktiota kutsutaan aina merkkijonon tutkimisen jälkeen. Valmis olio sijoitettiin TestDataControl.py-tiedostoon.

3.2.2 Testaus

Käyttöliittymäsovellus sisältää kolme oliota, joiden vuorovaikutuksesta ohjelma koostuu. Oliot on sijoitettu omiin lähdekoodeihin. Oliot testattiin yksitellen ennen yhteen liittämistä testaustarkoitukseen kirjoitetulla testialustaohjelmalla. Testausalustaohjelmaan importattiin tarvittavat moduulit sekä testattava olio. Testausalusta tulostaa standardiin ulostuloon tuloksia testin edetessä.

Käyttöliittymä testattiin importtaamalla käyttöliittymän lähdekoodi ja wx-moduuli testausalustaohjelmaan.

```
import Gui
import wx
```

KUVA 19. Moduulin importtaus Python-kielessä

Testausalustassa olion määritelmästä ja wx.App-määritelmästä luotiin oliot ja tämän jälkeen kutsuttiin wxApp-olion MainLoop()-funktiota, joka jää tutkimaan käyttöliittymän graafisten komponenttien aiheuttamia eventtejä. Auenneesta käyttöliittymästä klikattiin jokaista eventin aiheuttavaa komponenttia ja tutkittiin terminaaliin tulostuvaa debuggaustekstiä.

Sarjaliikenteen hallinta -olion testaus tehtiin ensin siten, että sarjaportin sijaan käytettiin tekstitiedostoa. Sarjaliikenteen hallinta -olio käyttää pySerial-moduulia, joka käsittelee sarjaportteja samalla tavalla kuin tiedostoja, joten kun olio saatiin toimimaan tekstitiedostojen kanssa, siihen tarvitsi muuttaa vain tekstitiedoston polun paikalle Linux-versiossa polku sarjaportissa olevaan laitteeseen ja Windows-versiossa käytettävän sarjaportin numero.

Olion testaus aloitettiin importtaamalla se testausalustaan. Työpöydälle luotiin tyhjä tekstitiedosto, joka simuloi sarjaporttia. Testausalustassa luotiin sarjaliikenteen hallinta -olion määritelmästä olio ja kutsuttiin olion sarjaportin avaavaa funktiota parametrilla ”path”, joka sisältää polun avattavaan tiedostoon. Testausalusta tulosti tiedon siitä, oliko sarjaportin tai tiedoston avaaminen onnistunut. Tämän jälkeen kutsuttiin olion funktiota, jolle parametrina annettava teksti kirjoitetaan sarjaporttiin tai tiedostoon. Työpöydällä oleva tekstitiedosto avattiin ja katsottiin löytyikö tiedostosta kirjoitettu teksti. Seuraavaksi testattiin sarjaportin lukemista. Testausalustasta kutsuttiin olion funktiota, joka lukee sarjaportista halutun määrän merkkejä. Funktiolle annetaan luettavien merkkien määrä parametrina, ja se palauttaa luettavat merkit sarjaportista tai tiedostosta. Funktion palauttamat merkit tulostettiin terminaaliin. Lopuksi testattiin sarjaportin tai tiedoston sulkemista. Testausalustasta kutsuttiin sarjaportin tai tiedoston sulkevaa olion funktiota. Tämän jälkeen testausalustassa tutkittiin onnistuiko sarjaportin tai tiedoston sulkeminen oikein, ja tulos tulostettiin terminaaliin.

4 TESTAUSJÄRJESTELMÄN ELEKTRONIIKAN SUUNNITTELU JA TESTAUS

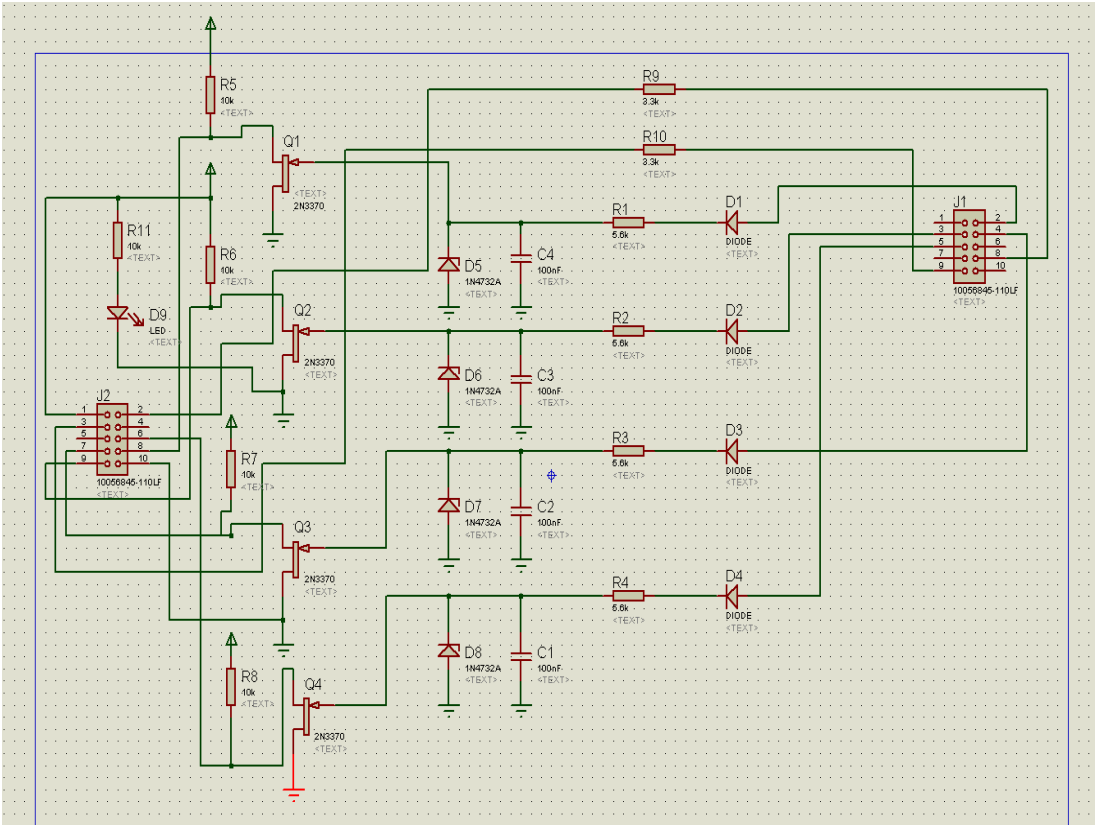
4.1 Elektroniikan suunnittelu

Elektroniikan osuus testausjärjestelmässä jäi kohtalaisen vähäiseksi. Elektroniikan tehtäviksi jäi muuttaa tehosignaalit logiikkatasoisiksi siten, että piirikortin mikrokontrolleri pystyy tulkitsemaan, toimiiko tehosignaalien muodostus spesifikaation mukaisella tavalla. Lisäksi elektroniikka sisältää pulssianturin liittimen testauksessa vaadittavat kaksi sarjavastusta varmistamaan, ettei oikosulkutilannetta pääse syntymään.

Molemmat tehosignaalit sisältävät kaksi kanavaa, joten testauselektroniikassa on neljä kanavaa tehosignaaleille. Signaalin muokkauksen periaatetta testattiin rakentamalla yksi kanava koekytkentälevylle. Kanavaan syötettiin signaaligeneraattorilla tehosignaalia mallintavaa signaalia ja mittaamalla ulos tulevan signaalin amplitudia oskilloskoopilla. Kun elektroniikan suunnittelun periaate oli testattu toimivaksi, rakennettiin varsinaisen testauselektroniikan prototyyppi.

4.1.1 Piirikaavion suunnittelu

Piirikaavio suunniteltiin Labcenter Electronicsin Proteus Isis -piirikaaviosuunnittelutyökalulla sekä simuloitiin ProSpice-simulointiohjelmalla. Piirikaavio (kuva 20) suunniteltiin siten, että tehosignaali lataa kondensaattoria C4 vastuksen R1 ja diodin D1 kautta. Kondensaattorin rinnalle on kytketty 4,7 voltin kynnysjännitteen omaava zener-diodi D5 ylijännitesuojaksi. Zener-diodin jälkeen kytkennässä on n-tyypin mosfet-transistori Q1, jonka hila on kytketty zener-diodin katodin kanssa yhteen. Mosfet-transistorin nielu on kytketty vastuksen R5 kautta käyttöjännitteeseen ja lähde on kytketty maahan. Transistorin nielu on kytketty mikrokontrollerin sisääntuloon J2. Jos transistorin hilalla vaikuttaa zener-diodin kynnysjännitteen suuruinen jännite, on transistori johtavassa tilassa ja nielu on yhteydessä maahan. Tällöin mikrokontrollerin sisääntulo on loogisessa 0-tilassa, joka ilmaisee tehosignaalin muodostuksen olevan toimiva. Mikäli tehosignaalin muodostuksessa on häiriö, kondensaattori ei lataudu ja transistori pysyy johtamattomassa tilassa. Tällöin transistorin nielulla ja samalla mikrokontrollerin sisääntulossa on käyttöjännitteen suuruinen jännite, eli ne ovat loogisessa 1-tilassa. Kuvassa 20 on esitetty testausjärjestelmän elektroniikan piirikaavio.



KUVA 20. Testausjärjestelmän elektroniikan piirikaavio

Tehosignaalin amplitudi on 25 volttia ja taajuus on 5 kHz. Signaali on kanttiaalto ja sen duty cycle on 50 prosenttia. Kondensaattorit C1-C4 latautuvat ajan (5kHz^-1)/2 eli 100 mikrosekuntia, minkä jälkeen ne purkautuvat zenerdiodien D5-D8 kautta maahan, kunnes kondensaattorien jännite on zenerdiodien kynnysjännitteen suuruinen. Fet-transistorien Q1-Q4 kautta ei kulje käytännössä lainkaan virtaa, joten kondensaattorit eivät purkaudu zenerdiodien kynnysjännitettä matalammaksi.

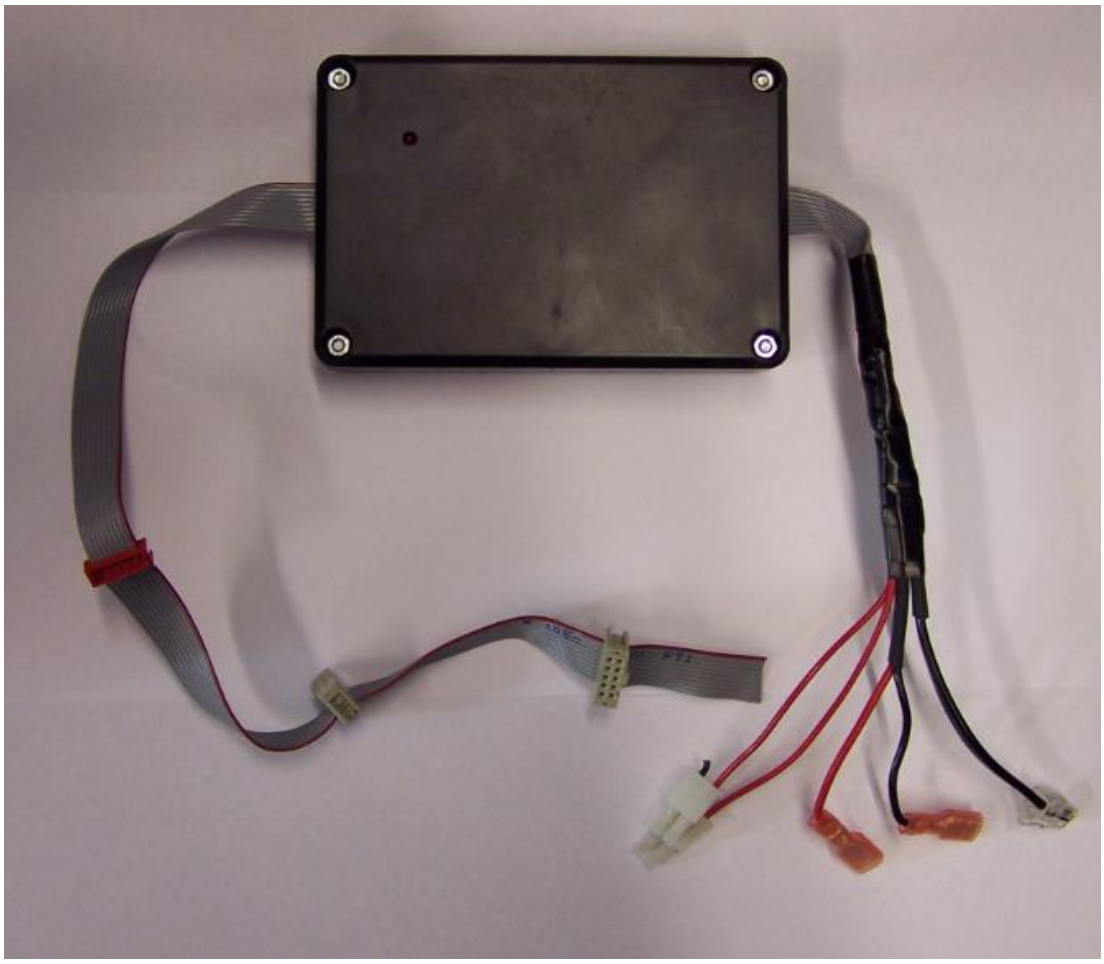
Kondensaattorin arvoksi valittiin 100nF ja vastuksen arvo laskettiin kaavasta 1. Tarvittavan vastuksen arvoksi saatiin 4802 ohm, joka pyöristettiin ylöspäin seuraavaan standardiin vastusarvoon 5,6 KOhmiin. Kondensaattorin halutaan siis latautuvat ensimmäisen tehosignaalin pulssin aikana arvoon 4,7 v, kun kondensaattorin arvo on 100 nF. Koska kondensaattori ei kuitenkaan purkaudu samalla nopeudella kuin se latautuu sarjadiodin estäessä purkautumisen tehosignaalin pulssin ollessa 0-tilassa, toisella tehosignaalin pulssilla kondensaattori latautuisi korkeampaan jännitteeseen kuin 4,7 volttia. Tämän takia kytkennässä on zener-diodi, joka tulee johtavaksi tehosignaalin toisella pulssilla pitäen kondensaattorin jännitteen halutussa 4,7 voltissa ja fet-transistorin johtavassa tilassa.

U(t) = Umax(1-e^(-t/RC)) *KAAVA 1*

4.1.2 Piirilevyn suunnittelu

Piirilevy suunniteltiin Labcenter Electronicsin Proteus Ares -piirilevysuunnittelutyökalulla. Koska testausjärjestelmän elektroniikka on kohtalaisen yksinkertainen ja vähän komponentteja sisältävä, sen prototyyppi suunniteltiin koottavaksi käsin aksiaalikomponenteista reikälevylle. Tämän takia piirilevysuunnitelma oli lähinnä ohje, miten kytkentä tullaan tekemään reikälevylle. Valmis prototyyppilevy suunniteltiin sijoitettavaksi elektroniikalle tarkoitettuun muovikoteloon. Tulevaisuudessa toimivaksi testattu testausjärjestelmän elektroniikka suunnitellaan pintaliitoskomponenteille.

Kun piirilevyn layout saatiin suunniteltua, komponentit juotettiin reikälevylle ja juotokset testattiin yleismittarin diodimittauksella. Piirilevyn muovikoteloon porattiin reikä piirikortin jännitteen saantia ilmaisevaa LED:iä varten ja tehtiin kolot lattakaapelien läpivienneille. Piirikortti asetettiin koteloon ja kotelo ruuvattiin kiinni.



KUVA 21. Testausjärjestelmän elektroniikan prototyyppi

4.2 Elektroniikan testaus

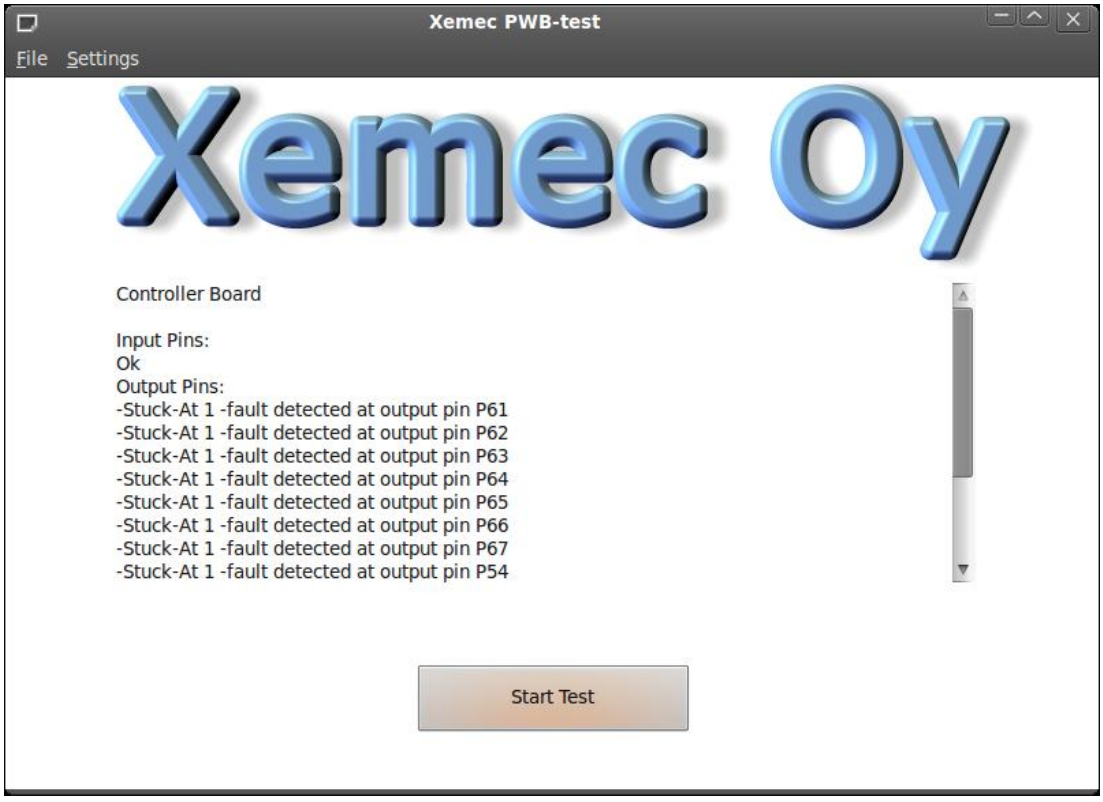
Prototyypin testauksen ensimmäisessä vaiheessa varmistettiin komponenttien ja liittimien juotokset yleismittarin diodimittauksella. Seuraavassa vaiheessa prototyypin neljä kanavaa testattiin yksitellen syöttämällä tehosignaaleja simuloivia signaaleja funktiogeneraattorilla kanaviin ja mittaamalla kanavien ulostuloja oskilloskoopilla.

Kun prototyypin kytkentä oli testattu funktiogeneraattorilla ja oskilloskoopilla toimivaksi, se kytkettiin testattavaan piirikorttiin. Piirikortin testiohjelmalle lähetettiin sarjaterminaalin kautta testinhallintakomento piirikortin tehosignaalien testaamiseksi ja tutkittiin testiohjelman lähettämät testitulokset. Vikatilanteita simuloitiin kytkemällä tehosignaaleja muokkaavien kanavien sisääntulot irti tehosignaalien liittimistä ja suoritettiin testi uudelleen. Tehosignaalien testaus toimi halutulla tavalla.

5 JÄRJESTELMÄN TESTAUS

Kun testausjärjestelmän kaikki osiot oli saatu valmiiksi ja testattua erikseen, ne liitettiin yhteen ja testausjärjestelmällä suoritettiin ensimmäinen vaatimusmäärittelyn mukainen testi piirikortille. Koska ohjelmiston osiot oli suunniteltu siten, että ne pystyttiin testaamaan itsenäisesti sellaisenaan, järjestelmä toimi halutulla tavalla ensimmäisestä testistä lähtien.

Vikatilanteita simuloitiin ensin irrottamalla lattakaapeliliittimiä piirikortista ensin yksi ja lopuksi useampi liitin kerrallaan. Käyttöliittymä tulosti irrotettua lattakaapeliliitintä vastaavien mikrokontrollerin pinnien nimet ja havaitun vian tyypin. Lopuksi vikatilanteita simuloitiin testausta varten valmistetulla lattakaapelilla. Siinä jokaiseen kaapelin johtimeen oli sijoitettu kytkin, jolla johdin saatiin johtamattomaan tilaan. Testausjärjestelmä toimi täysin spesifikaatioiden mukaisella tavalla.



KUVA 22. Käyttöliittymän tuloste vikoja havainneen testin jälkeen

Kun testausjärjestelmä oli testattu toimivaksi, sillä suoritettiin 30 piirikortin sarjan testaus. Tässä vaiheessa haluttiin tutkia lähinnä testausjärjestelmän nopeutta ja käytettävyyttä.

Ennen testauksen aloittamista käyttöliittymäsovelluksesta määritettiin tiedosto testiraportin tallennusta varten. Kuvassa 23 on esitetty osa 30 piirikortin sarjan testiraportista.

```
Test 21
-----------------------------------------------
Controller Board

Input Pins: Ok
Output Pins: Ok
Can Function: Ok
Mod4 Pins: Ok
Motor Interface: Ok


Test 22
-----------------------------------------------
Controller Board

Input Pins: Ok
Output Pins:
-Stuck-At 1 -fault detected at output pin P61
Can Function: Ok
Mod4 Pins: Ok


Test 23
-----------------------------------------------
Controller Board

Input Pins: Ok
Output Pins: Ok
Can Function: Ok
Mod4 Pins: Ok
Motor Interface: Ok
```

KUVA 23. Näyte testiraportista

Piirikorttisarjan testaukseen kului kokonaisuudessaan aikaa noin 25 minuuttia. Tähän sisältyivät testiohjelman ohjelmointi piirikorttien mikrokontrollerien Flash-muistiin, raporttitiedoston valitseminen, testauselektroniikan kytkeminen ja piirikorttien testaus.

6 JATKOKEHITYSMAHDOLLISUUDET

6.1 Testausjärjestelmän suunnittelua rajoittavat tekijät

Testausjärjestelmän suunnittelu aloitettiin, kun automaatiolaitteen piirikorttien layout oli suunniteltu jo valmiiksi. Layoutia ei voitu enää muuttaa joten testausjärjestelmä piti suunnitella vallitsevien olosuhteiden aiheuttamien rajoitteiden puitteissa.

Tästä johtuen mikrokontrollereille suunniteltiin ohjelma, joka simuloi ja tutkii piirikortin normaalia toimintaa. Piirikortin rajapinta muuhun järjestelmään on sen lopulliseen käyttötarkoitukseen suunnitellut I/O-liittimet, joita testausjärjestelmä hyödyntää. Testiohjelman lataaminen ja testauselektroniikan sekä sarjaliittimen kytkeminen piirikortin I/O-liittimiin on hidasta ja vie suurimman osan testaukseen kuluvasta ajasta. CAN-väylän normaalia toimintaa ei myöskään voida testata ilman toista CAN-väylää käyttävää laitetta, joten CAN-väylästä testataan ainoastaan mikrokontrollerin sisältämän CAN-prosessorin toiminta.

6.2 Testausjärjestelmän jatkokehitys

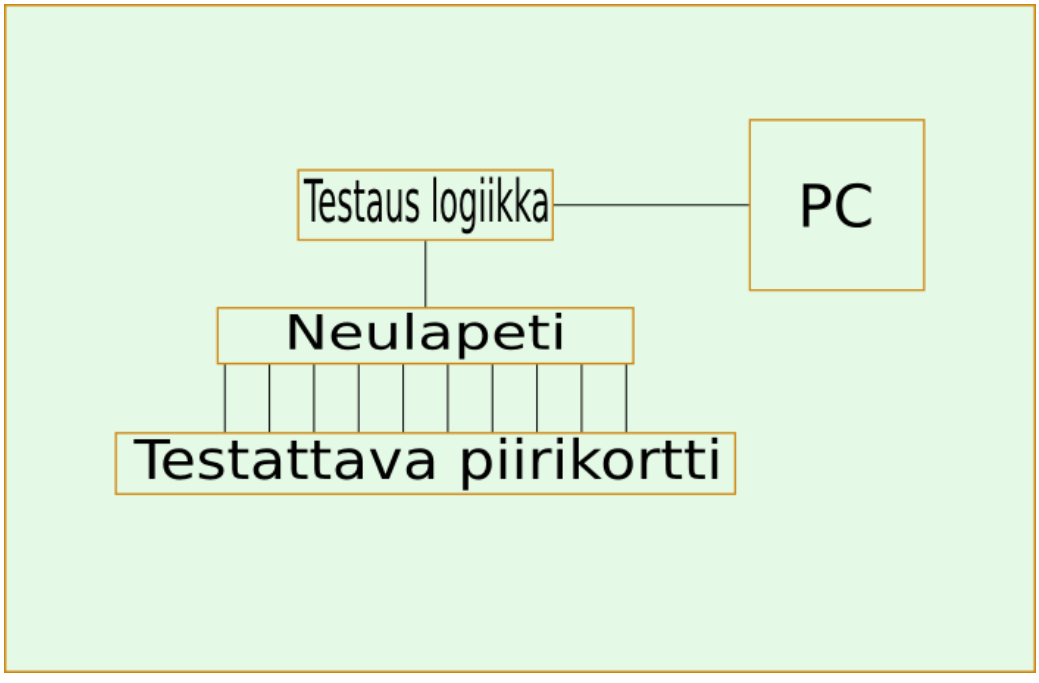
6.2.1 Seuraavat muutokset

Testauselektroniikkaan tullaan liittämään CAN-väylää käyttävä mikrokontrolleri, jotta väylä voidaan testata kokonaisuudessaan toimivaksi. Testauselektroniikan mikrokontrollerille tullaan suunnittelemaan ohjelma, joka vastaanottaa testattavalta piirikortilta viestejä ja lähettää kuittauksen takaisin, mikäli vastaanotettu viesti oli vaaditunlainen. Myös piirikorttien testausohjelman CAN-väylän testauksen suorittava osio tullaan suunnittelemaan edellä mainitun takia uusiksi. Nämä muutokset tullaan toteuttamaan samalla, kun piirikortin layout suunnitellaan pintaliitoskomponenteille.

6.2.2 Pidemmän tähtäimen muutokset

Automaatiolaitteen piirikorttien seuraavan sukupolven yhteydessä tullaan tutkimaan mahdollisuutta rakentaa testausjärjestelmä neulapetitestaukselle. Piirikortit vaativat tässä tapauksessa erillisiä testipadeja, joista neulapedin neulat ottavat kontaktin testattaviin signaaleihin.

Suunnitteilla olevan tyyppisessä neulapetitestauksessa piirikorteille ei tarvitse ladata erillistä testiohjelmaa, koska juotokset testataan käyttöjännitteiden ollessa pois kytkettyinä. Myöskään erillisiä liittimiä ei tarvitse kytkeä, sillä kaikki signaalit tulevat yhden, piirikortin päälle asetettavat neulapedin kautta. Nämä seikat nopeuttaisivat piirikorttien testausta huomattavasti. Haittapuolena tällaisessa testauksessa on se, että se ei tutki mikropiirien toimivuutta vaan ainoastaan piirilevyn juotokset. Kuvassa 24 on esitetty periaatekuva neulapetitestauksesta.



KUVA 24. Neulapetitestauksen periaate

7 POHDINTA

Opinnäytetyön tekeminen oli erinomainen tapa tutustua tilaavan yrityksen käytössä oleviin teknologioihin ja menetelmiin. Aihe oli mielenkiintoinen sekä sopivan laaja ja vaativa.

Opinnäytetyön aihealue oli pääsääntöisesti ohjelmiston suunnittelua, vaikka suuntautumisvaihtoehtoni opinnoissa on elektroniikkapainotteinen. Tämä seikka toi haastavuutta ja mielenkiintoa opinnäytetyöprojektiin. Aikaisempi kokemus sekä mielenkiinto sulautettujen järjestelmien ohjelmointiin helpotti ohjelmiston kehitystä.

Vaikka varsinainen testausjärjestelmä on pääsääntöisesti ohjelmistoa, piti testattavaan elektroniikkaan tutustua syvällisesti, jotta testit voitaisiin suunnitella oikean tyyppisiksi. Testiherätteitä sekä järjestelmän vaatimaa lisäelektroniikkaa suunniteltaessa pääsin hyödyntämään myös elektroniikan osaamistani.

Testausjärjestelmän käyttöliittymäsovellus ohjelmoitiin Python-ohjelmointikielellä, joka oli projektin alkuvaiheessa minulle täysin tuntematon, joten projektin aikana pääsin opettelemaan kokonaan uuden ohjelmointikielen. Kirjoitushetkellä olen Pythonin kanssa tekemisissä lähes päivittäin, joten sen opetteleminen oli erittäin hyödyllistä projektin aikana.

Tilaavalle yritykselle testausjärjestelmästä on runsaasti hyötyä. Järjestelmä nopeuttaa piirikorttien testausta arviolta 20-kertaiseksi, joten järjestelmän käytöstä kertyy pitkällä aikavälillä huomattavia rahallisia säästöjä.

LÄHTEET

A Brief History of the C Language 2010. Saatavissa: http://hubpages.com/hub/A-Brief-History-of-the-C-Language. Hakupäivä: 1.7.2010.

Silander, Simo 1999. Ohjelmoinnin perusteet ja C-kieli, modulaarinen ohjelmointi. Saatavissa: http://cs.stadia.fi/~silander/ohjelmointi/c_opas-Modulaar.html. Hakupäivä: 1.7.2010.

Tapaninen, Timo 2007. Tietojenkäsittelytieteen historia –seminaari, kevät 2007.
Saatavissa: http://www.cs.helsinki.fi/u/kerola/tkhist/k2007/alustukset/ohjparadigma/ATT00051.pdf. Hakupäivä 1.7.2010.

A Brief History of Python 2010. Saatavissa: http://python.about.com/od/gettingstarted/ss/whatispython_2.htm. Hakupäivä: 1.7.2010.

What is wxPython? 2010 Saatavissa: http://www.wxpython.org/what.php. Hakupäivä: 1.7.2010.

Mercurial. Saatavissa: http://linux.fi/wiki/Mercurial. Hakupäivä: 1.7.2010.